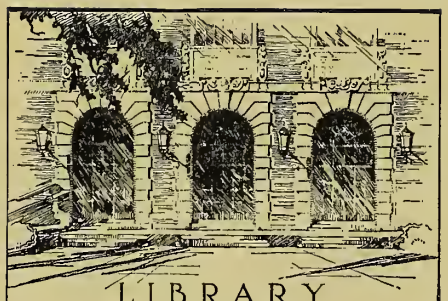
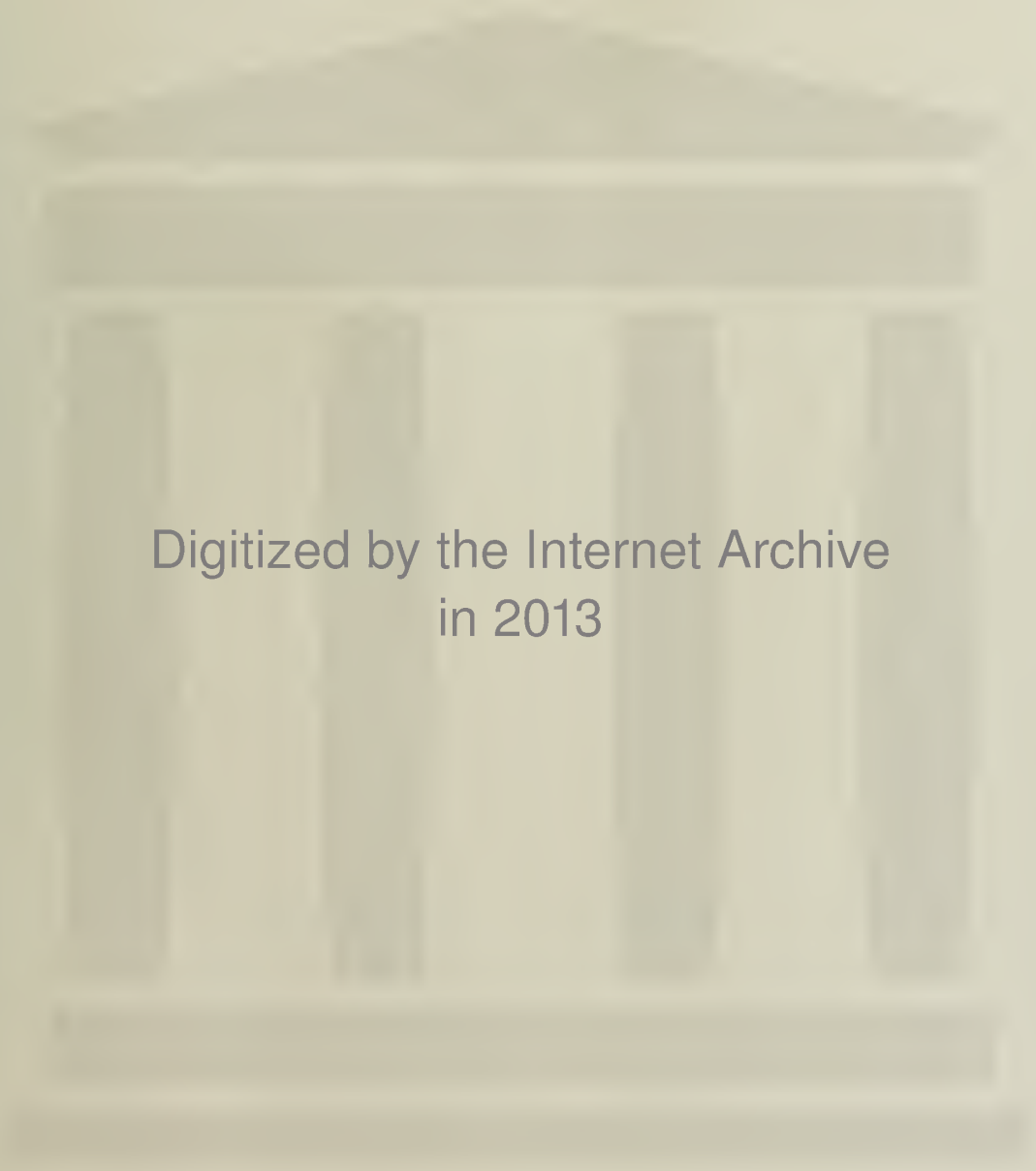




DIGITAL COMPUTER LABORATORY
GRADUATE COLLEGE
UNIVERSITY OF ILLINOIS

User's Manual
for the
ALCØR-ILLINØIS-7090
ALGØL-60 Translator
University of Illinois
2nd Edition

Manual Written and Revised by
R. Bayer E. Murphree, Jr. D. Gries

September 28, 1964

Preface

In June, 1962, by arrangement between the University of Illinois and the ALCOR group in Europe, Dr. Manfred Paul, Johannes Gutenberg Universität, Mainz, and Dr. Rüdiger Wiehle, München Technische Hochschule, Munich, began the design of the ALCØR-ILLINØIS-7090 ALGOL-60 Translator, the use of which is described in this Manual. They were joined in July, 1962, by David Gries and the writer and later by Michael Rossin, Theresa Wang and Rudolf Bayer, all graduate students at the University of Illinois.

This manual is an effort to explain the differences which exist between publication ALGOL-60 as it is defined by the Revised Report on the Algorithmic Language ALGOL-60 (as published in the January 1963 issue of Communications of the Association for Computing Machinery) and as it actually has been implemented by the group named above. Relatively few features of ALGOL-60 have not been implemented, so this manual consists mostly of an explanation of the "hardware" representation of true ALGOL rather than deviations from it.

The translator itself has been completed, and it has been running for some time on the University of Illinois 7094/1401 computer system. Convenience features such as input/output routines and debugging aids are continually being developed and as they are completed, documentation and instructions for their use will be available as parts of this manual.

In the preparation of this manual, the writer is particularly indebted to John Moore, Research Assistant at the DCL, for his critical reading of the manuscript and his many suggestions for its improvement.

Section 9 was prepared in its entirety by David Gries during his stay at München Technische Hochschule.

E. L. Murphree, Jr.
June 1, 1964

User's Manual for the ALCØR-ILLINØIS-7090 ALGOL-60 Translator, 2nd Edition

Contents

1. Introduction

The programming language ALGOL-60 (hereafter called simply ALGOL) provides a very versatile means of expressing a certain class of algorithms in a form which is at the same time palatable for human beings and sufficiently precise for automatic translation by computing machines. The designers of ALGOL set out to define just such an artificial language and chose basic symbols that would meet these aims without regard to the character set available.

As a result of this disregard for the actual availability of the basic symbols of ALGOL, a relatively large number of them are not currently available in the symbol sets for commercial computers. Clearly this is true for the word symbols, go to, if, while, etc., but it is also true for such commonly encountered mathematical symbols as × (multiplication sign), ÷ (integer divide symbol), <, ≤, ≥, >,], [, and even true, in most cases, for the lower case alphabet. Since 1958, when the first ALGOL Report appeared, no serious effort has been exhibited by the computer manufacturing industry to fill the need by producing special ALGOL character sets; hence, in order to make use of this powerful programming tool ALGOL certain alterations have been proposed for the ALGOL symbol set when the symbols are a part of a program which is to be entered into a computer.

This alternate set of ALGOL symbols is called the "hardware representation," and is more or less standardized by SHARE* in the United States and the ALCØR-Group* in Europe. But for this difference between publication ALGOL and "hardware ALGOL," the major portion of this manual would be unnecessary. This difference does, however, exist, and one of the aims of this manual is to make it possible for the user to run programs written in ALGOL on a 7090/94 computer, using the ALCØR-ILLINØIS-7090 ALGOL Translator (hereafter called the Translator). This Translator was

---

*SHARE is the IBM 700/7000 series user's group.
The ALCØR-Group is a European user's group organized around existing ALGØL compilers.

written so that it could be run under the control of the PORTHOS Monitor System (also written at the University of Illinois), and certain requirements imposed on each job by PORTHOS are stated in a later section. It is intended, however, that this manual be largely independent of system requirements, so that it can be used at any 7090/94 installation which has access to the Translator. Hence, monitor requirements are clearly identified and can be disregarded by those who are using the Translator independently of PORTHOS.

In writing this manual, it has been assumed that the users will be divided roughly into two groups: those individuals who have a working knowledge of ALGOL and merely want information to enable them to transform their programs into hardware form and get the programs onto the computer; and those persons who know nothing about ALGOL, but know how to program in some other language and wish to extend their programming activities to include the use of ALGOL. To the former, the manual text has been addressed; for the latter, Appendix A has been provided, in the form of a bibliography of tutorial papers and books on ALGOL with a brief discussion of each. These individuals should turn to Appendix A now, and select reading material in order to learn ALGOL. Further study of this Manual without some knowledge of the language would probably prove fruitless. The list is not exhaustive. For other and more recent publications in English on the subject, the reader is referred to publications of the Association for Computing Machinery and The British Computer Society. There are, of course, other publications in which articles on ALGOL appear. In addition, a copy of the ALGOL Report has been included as Appendix B.

It is important to note that the prospective user is expected to already know how to program for a digital computer. This manual is in no way to be regarded as a primer in programming, in ALGOL or any other programming language.

The form of presentation has been chosen because of its inherent flexibility. There will undoubtedly be revisions in one form or another of parts of this manual. In particular, changes and additions can be expected in the section devoted to input/output, as new and revised procedures for input/output become available.

Lastly, it should be noted that this manual is a user's manual, and contains little information for the relatively rare individual who feels that he must alter the translating program itself.

The general arrangement of the presentation is from the simplest and most basic requirements, through input/output conventions, to the implementation of "code" procedures, and debugging aids. Suggestions from readers for improving the usefulness of this manual will be welcome.

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." Every installation which uses the Translator has the option of permitting the use of certain abbreviated representations of ALGOL symbols by changing a parameter in the Translator. 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. Whether such a use is a fatal error depends upon the policy of the installation. All representations are acceptable at the University of Illinois. 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 'GØ TØ'. It should be noted that except in strings, blanks are ignored in hardware representation, just as in publication ALGOL, so that 'GØ TØ' is identical to 'GØTØ' or 'GØ TØ' and 'BE GIN' is as acceptable as 'BEGIN'.

Minor restrictions not mentioned in the ALGOL Report have been imposed by the nature of the 7090/94 computer. The word size of the 7090/94 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:

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

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

Date:	6/1/64
Section:	2.2.
Page:	1 of 1
Change:	1

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 representations is given below.

The careful reader will note that two non-ALGOL word symbols, 'CØDE' 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	'GØ TØ'
if	'IF'
then	'THEN'
else	'ELSE'
for	'FØR'
do	'DØ'
step	'STEP'
until	'UNTIL'
comment	'CØMMENT'
begin	'BEGIN'
end	'END'
own	'ØWN' (See section 8.1.)
Boolean	'BØØLEAN'
integer	'INTEGER'
real	'REAL'
array	'ARRAY'
switch	'SWITCH'
procedure	'PRØCEDURE'
string	'STRING'
label	'LABEL'
value	'VALUE'
code	'CØDE'
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 7090/94 and 1401 character sets. The hardware representations of all the arithmetic operators are tabulated below.

ALGOL Symbol	Description	Hardware Representation	Tolerated Representation
+	plus	+	
-	minus	-	
×	multiplication	*	
/	division	/	
÷	integer devision	//	
↑	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 at many installations. 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	'NØT GREATER'	'LQ'
$=$	equal to	'EQUAL'	'EQ'
$\geq$	greater than or equal to	'NØT LESS'	'GQ'
$>$	greater than	'GREATER'	'GR'
$\neq$	not equal to	'NØT EQUAL'	'NQ'
$\equiv$	logical equivalent	'EQUIV'	'EQV'
$\supset$	logical implies	'IMPL'	'IMP'
$\vee$	logical or	'ØR'	
$\wedge$	logical and	'AND'	
$\neg$	logical negation	'NØT'	

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.

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 'CØDE' Word Symbol.

The 'CØDE' 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 'CØDE'.

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 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 index 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 names of arrays. 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> ::= FØRMAT (<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

FØRMAT (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	B1 (input) or A3 (output)	regular input (output) tape
2	B2	scratch tape
3	B3	scratch tape
4	A4	scratch tape
5*	B4	scratch tape
6	A3	regular print tape
7	B1	regular input tape
9	B5	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 (B1) if the call is readf or readmatrixf or the regular print tape (A3) if the call is printf or printmatrixf.

<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.

*using logical tape 5 in connection with an output procedure causes information written on the output tape with an indication that it is to be punched rather than printed.

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,

FØRMAT (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

FØRMAT (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,

FØRMAT (7, "F 10.4, 5 X, F 10.4")

and

FØRMAT (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, 133* 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.

*the first character of every output record to be printed is used as carriage control by the 1401. So at most 132 characters per line can actually be printed. For information about carriage control see the PØRTHØS-manual.

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 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 $\pm$. Note that every number in this conversion has at least six columns of its field used for "bookkeeping" symbols:

$\pm$. E $\pm$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).

Deviations from this form are possible for input and are described in the PORTHOS Manual. No deviations are permitted for output.

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 be 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,

```
FØRMAT (7, "23 HbCØMPUTEDbAVERAGESbb=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

bCØMPUTEDbAVERAGESbb=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.

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. 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 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>)

<printmatrixf call> ::= PRINTMATRIXF (<array identifier>)

<output list> ::= <arithmetic expression> | <output list>,
<arithmetic expression>

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 expressions. 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 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 3.4.3.), 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 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.

3.7. Data.

The actual introduction of data into the computer is done by means of the PORTHOS Monitor System (see section 5), and requires the use of one basic PORTHOS control instruction

$ DATA

which must appear on a card, with the $ in column 1, which immediately follows 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. Deviations from the established formats are discussed in the PORTHOS Manual.

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.

.3. Procedures Written in Other Source Languages.

This type of procedure can easily be incorporated into an object program produced by the ALGOL translator. However, the user must write such a procedure so as to be compatible with the translator-produced object program. All information necessary to do this will be found in Section 9.

5. The PORTHOS System and the ALGOL Translator.

The Translator was originally designed to be run under the control of the PORTHOS Executive 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. There are, of course, other versions of the Translator which have had the necessary routines added as a package so that use can proceed independently of PORTHOS. Since the PORTHOS system is well documented in a user's manual, the details will not be repeated here. Instead, a description of those requirements imposed by PORTHOS which are most evident to the user will be discussed.

The 7094/1401 computer system at the Digital Computer Laboratory, University of Illinois, 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 Digital Computer Laboratory 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 computers, and processed one after the other by the 7094. Every job must, then, be appropriately identified with a program card. This card is described in the PORTHOS user's manual. Since the system must be informed what processing is to be carried out on the job following the program card, $ control cards must be placed before that job. These cards are completely covered in the PORTHOS user's manual. The minimum control cards for translation and execution of an ALGOL program are

$ ALGØL
$ GØ

and if the program contains data, then the data must be preceded by a

$ DATA

card. The inclusion in a job of column binary cards is covered in the section in this manual on Procedures.

Because of the construction of the Translator, certain of the $ control cards described in the PORTHOS manual are meaningless when submitted with an ALGOL program. These are

```
$ SCATRE
$ MAD
$ FØRTRAN
$ PRINT ØBJECT
```

The first three are obviously meaningless because they call other translators; the last is meaningless because during the translation process, ALGOL programs do not go through a symbolic object code phase.

A discussion of the use of the system library routines appears in the section of this manual on Procedures.

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 PORTHOS control cards (See Section 5.), and properly submitted at the dispatching room, 110A ERL. The subjects discussed in the following sections are more fully developed in the PORTHOS User's Manual, but the essential details are covered here.

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' VØLUME, PRESSURE, TEMPERATURE., 'INT
EGER' NUMBER, EXPERIMENT, DATE., 'REAL' 'PRØCEDURE'
INTEGRATION (ALPHA, BETA, GAMMA).,
```

is as acceptable and meaningful as

```
'BEGIN'
    'REAL' VØLUME, PRESSURE, TEMPERATURE.,
    'INTEGER' NUMBER, EXPERIMENT, DATE.,
    'REAL' 'PRØCEDURE'
        INTEGRATIØN (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 comment word symbol and comments after the final end of the programs. These

ALGOL comments have nothing to do with system $$ comment cards, which are addressed to the computer operator (see PORTHOS Manual).

6.3. System Control Cards for ALGOL.

The minimum PORTHOS system control cards required for translation and execution of an ALGOL source program with data cards are

```
$ ALGØL
$ GØ
$ DATA
```

Because of the unique dynamic storage feature of ALGOL, the other applicable PORTHOS system control cards do not cause identical action with ALGOL as with other programming systems. A full description of these control cards and their action in general appears in the PORTHOS Manual; their unique actions with ALGOL are described in Section 5.

The control cards $ ALGØL and $ GØ respectively cause translation of the ALGOL source program into machine code and execution of the machine code program.

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.). In the event that the data card format is not identical to that specified by the applicable format call, then the data may be read, with the actual format overriding the specified format. For fuller details, see the PORTHOS user's manual.

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 precompiled (preassembled) subroutines.

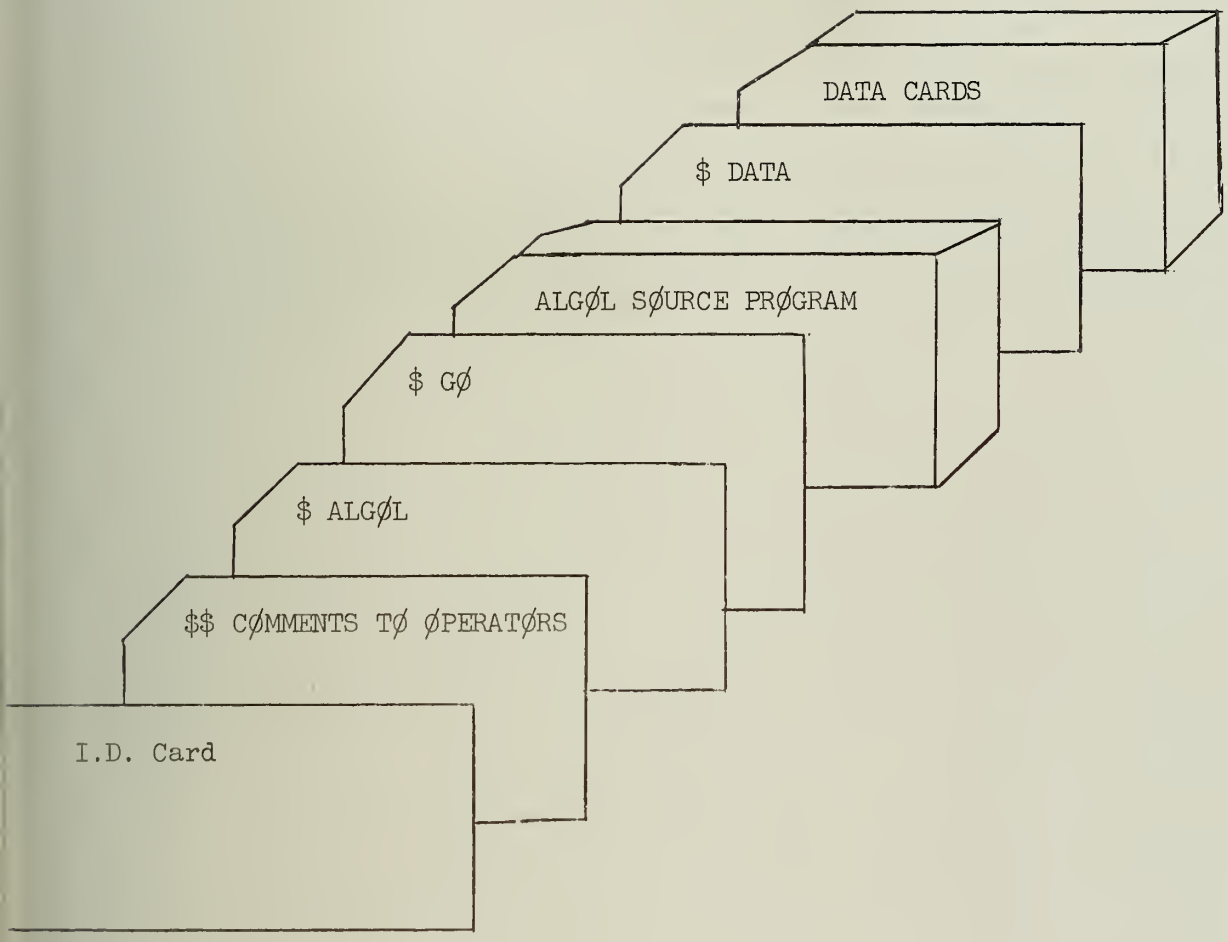
If a user gets, by use of the system control card $ PUNCH ØBJECT, a binary deck of his translated object program, then he may use this binary deck whenever he chooses without further reference to ALGOL.

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 9.

6.6. Library Routines on Tape or Cards.

The subroutine library is at the disposal of the ALGOL user through the code procedure declaration. The user has to make sure, however, that the proper linkage between an Algol program and the used library subroutine is provided. This can be done e.g. by a SCATRE subroutine which makes up the right linkage and then calls the library subroutines. For details see section 9.

6.7. Typical Deck Makeup.



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

if (Boolean Expression) then Expression E_1; else 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 end in a multiple-block program, the translator normally has insufficient information on which to base an attempt at inserting the missing end's, and so must abandon its attempt at translation. That is, a sequence of begin'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            begin  end
               end            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;
          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.

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 machines 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.).

There may be, in addition to ALGOL error messages, system error messages; the latter are not discussed here, and the user is directed to the PORTHOS User's Manual for explanation of those error messages.

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.

Following are the error messages generated by the Translator. The symbol ... indicates that the Translator inserts a variable name or a number unique to the program being translated. Figures 7.1 and 7.2 show typical output for erroneous programs.

```
ALGOR- ILLINOIS 7090 ALGOL COMPILER                                    PAGE 01

  1      'BEGIN'   'REAL'  'PROCEDURE' B(X)., 'INTEGER' X.,   'CODE' .,
  2              'REAL'  C.,
  3      'PROCEDURE'  Y(X)., 'CODE'.,
  4      M(3).,   Y(LABEL).,
  5      LABEL..  C.= B(3).,
  6
  7      'BEGIN'   'REAL'  AZ.,
  8                'READ' 'ARRAY'  A(/1.,2.,3.,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 OCCURENCE 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

```
ALGOR- ILLINOIS 7090 ALGOL COMPILER                                    PAGE 01

  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 CAREFULLY

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

SUCCESSFUL COMPILATION, BUT PROGRAM IS TOO LARGE TO FIT IN MEMORY.

IDENTIFIER...WITH KIND...IS NOT DESIGNATIONAL.

THE SWITCH...IS DECLARED TWICE.

THE STRING WAS USED BEFORE WITH ANOTHER KIND.

DECLARATION AFTER THE STATEMENT OR LABEL.

PROCEDURE--FORMAL PARAMETER PART...WAS USED BEFORE WITH A DIFFERENT NUMBER OF SUBSCRIPTS.

TYPE OF LEFT SIDE OF .= DIFFERS FROM RIGHT SIDE.

ERROR IN ALGOL COMPILER.

...FOLLOWING A DESIGNATIONAL 'THEN'

...FOLLOWING AN ARITHMETIC 'THEN'

...FOLLOWING A STATEMENT 'THEN'

THE EXPRESSION BETWEEN 'THEN' AND 'ELSE' SHOULD BE BOOLEAN.

...FOLLOWING A DESIGNATIONAL 'THEN'

THE EXPRESSION BETWEEN 'THEN' AND 'ELSE' SHOULD BE DESIGNATIONAL.

THE EXPRESSION BETWEEN 'THEN' AND 'ELSE' SHOULD BE ARITHMETIC.

...FOLLOWING A NON-ARITHMETIC 'ELSE'.

...FOLLOWING A STATEMENT 'ELSE'.

UNDETERMINED EXPRESSION IS FOLLOWED BY...

...FOLLOWING A 'THEN' OF A BOOLEAN OR ARITHMETIC EXPRESSION.

...FOLLOWING A 'THEN' OF AN UNDETERMINED EXPRESSION. (WHICH CAN ONLY OCCUR ON AN ACTUAL PARAMETER POSITION).

...FOLLOWING A 'THEN' OF A BOOLEAN EXPRESSION.

...FOLLOWING A 'THEN' OF A CONDITIONAL EXPRESSION.

LABEL...IS DECLARED TWICE.

THE EXPRESSION BETWEEN 'IF' AND 'THEN' IS NOT BOOLEAN.

FOR LIST IS WRONG. NO. OF 'STEP'S IS DIFFERENT FROM NO. OF 'UNTIL'S.

FOR LIST IS WRONG. 'WHILE' IS SYNTACTICALLY INCORRECT.

FOR LIST IS WRONG. 'UNTIL' IS SYNTACTICALLY INCORRECT.

ARRAY DECLARATION...WAS USED BEFORE WITH A DIFFERENT NO. OF SUBSCRIPTS.

ARRAY DECLARATION...WAS USED BEFORE WITH TYPE BOOLEAN.

ARRAY DECLARATION...WAS USED BEFORE WITH TYPE INTEGER.

ARRAY DECLARATION...WAS USED BEFORE WITH TYPE 'REAL'.

ARRAY DECLARATION...WAS USED BEFORE WITH A DIFFERENT KIND.

FOR LIST IS WRONG. TOO MANY 'STEP' DELIMITERS.

ARRAY DECLARATION...AN UPPER OR LOWER BOUND HAS BEEN LEFT OUT.

DIFFERENT TYPES IN THE LEFT PART LIST OF THIS ASSIGNMENT STATEMENT.

IDENTIFIER...WITH KIND...WAS USED BEFORE WITH A DIFFERENT NO. OF SUBSCRIPTS.

ARRAY DECLARATION. BOUND PAIR SEPARATORS NOT RIGHT.

A SUB-EXPRESSION IN AN EXPRESSION SHOULD BE BOOLEAN.

A SUB-EXPRESSION IN AN EXPRESSION SHOULD BE ARITHMETIC.

IDENTIFIER...DECLARED AS...WAS USED BEFORE WITH A DIFFERENT KIND.

A SEQUENCE OF DIGITS IS FOLLOWED BY THE LETTER...

ILLEGAL CONSTANT. AN INTEGER IT IS TOO LARGE. THE CHARACTER FOLLOWING IS...

PROGRAM HAS TOO MANY FOR LOOPS.

ILLEGAL CONSTANT. IT IS TOO LARGE. THE CHARACTER FOLLOWING IS...

ILLEGAL CONSTANT. IT IS NO ALGOL NO. THE CHARACTER FOLLOWING IS...

ILLEGAL CONSTANT. AN EXPONENT IT IS TOO LARGE. THE CHARACTER FOLLOWING IS...

ILLEGAL CONSTANT. NO DIGIT AFTER EXPONENT SIGN. THE CHARACTER FOLLOWING IS...

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

...FOLLOWED BY A SEQUENCE OF DIGITS FOLLOWED BY...

IDENTIFIER...IS DECLARED TWICE.

IDENTIFIER...IS A FORMAL PARAMETER IN A SURROUNDING BLOCK.

IDENTIFIER...IS NOT FUNCTION NAME IN ACTUAL BLOCK.

IDENTIFIER...IS A FORMAL PARAMETER IN THIS BLOCK.

IDENTIFIER...IS NOT FORMAL PARAMETER IN ACTUAL BLOCK.

IDENTIFIER...IS NOT IN A SURROUNDING BLOCK.

IDENTIFIER...IS NOT ARITHMETIC VARIABLE.

IDENTIFIER...IS NOT SIMPLE OR FUNCTION NAME.

IDENTIFIER...IS NOT ARRAY OR FUNCTION WITH PARAMETERS.

IDENTIFIER...IS NOT DECLARED OR IS UNDEFINED AT THIS POINT.

IDENTIFIER...IS NOT BOOLEAN ARRAY.

IDENTIFIER...IS NOT BOOLEAN FUNCTION WITH PARAMETERS OR ARRAY.

IDENTIFIER...IS NOT LABEL.

IDENTIFIER...IS NOT SWITCH.

IDENTIFIER...IS NOT ARRAY.

TYPE OF...SHOULD BE BOOLEAN. TYPE OF THE EXPRESSION BEFORE...SHOULD BE BOOLEAN.

IDENTIFIER...IS NOT SIMPLE OR FUNCTION WITH PARAMETERS.

TYPE OF...SHOULD BE INTEGER TYPE OF THE EXPRESSION BEFORE...SHOULD BE INTEGER.

TYPE OF...SHOULD BE ARITHMETIC TYPE OF THE EXPRESSION BEFORE...SHOULD BE ARITHMETIC.

IDENTIFIER...IS NOT ARRAY OR PROCEDURE WITH PARAMETERS.

IDENTIFIER...IS NOT PROCEDURE WITHOUT PARAMETERS.

IDENTIFIER...IS NOT PROCEDURE WITH PARAMETERS.

FORMAL PARAMETER...CANNOT APPEAR ON LEFT SIDE OF .= SIGN.

IDENTIFIER...IS NO ALGOL IDENTIFIER. ERROR IN LGL ROUTINE...

FORMAL PAR. CORRESP. TO ACTUAL PAR...CANNOT APPEAR ON LEFT SIDE OF .= SIGN.

CALL OF FUNCTION OR PROCEDURE. ACT. PAR. (...) NOT COMPATIBLE WITH FORMAL PAR. (...).

CALL OF FUNCTION OR PROCEDURE. THE NUMBER OF PARAMETERS IN...WITH KIND... DIFFERS FROM PREVIOUS USE.

DECLARATION OF FUNCTION OR PROCEDURE. PARAMETER DELIMITER IS WRONG-- (COLON MISSING).

DECLARATION OF FUNCTION OR PROCEDURE. PARAMETER DELIMITER IS WRONG. (TWO COLONS).

DECLARATION OF FUNCTION OR PROCEDURE. PARAMETER DELIMITER IS WRONG-- (LETTER FOLLOWING A COLON).

DECLARATION OF FUNCTION OR PROCEDURE. SEMICOLON MISSING AFTER 'CODE'.

DECLARATION OF FUNCTION OR PROCEDURE. TYPE OF...WITH KIND...DIFFERS FROM PREVIOUS USE.

DECLARATION OF FUNCTION OR PROCEDURE...WITH KIND...DIFFERS FROM PREVIOUS USE.

DECLARATION OF FUNCTION OR PROCEDURE. THE NO. OF PARAMETERS IN...WITH KIND... DIFFERS FROM PREVIOUS USE.

DECLARATION OF FUNCTION OR PROCEDURE. IDENTIFIER...WAS USED BEFORE WITH A DIFFERENT KIND.

CALL OF FUNCTION OR PROCEDURE. ACT. PAR. (...) NOT COMPATIBLE WITH FORMAL PAR. (...).

THERE ARE...MORE 'BEGIN'S THAN 'END'S.

THERE ARE MORE 'END'S THAN 'BEGIN'S.

CHAR. ...APPEARED IN COMMENT AFTER 'END'. PERHAPS A ; IS MISSING.
TRANSLATION NOT STOPPED.

IDENTIFIER...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 CAREFULLY.

NO 'FINIS' AT END OF PROGRAM.

THE LONG IDENTIFIER IS TOO LONG.

SYNTAX CHECKER STOPPED. PROGRAM HAS TOO MANY UNDECLARED IDENTIFIERS.

THE KIND OF FORMAL PARAMETER...OF PROCEDURE OR FUNCTION...CANNOT BE DETERMINED BY COMPILER. IT WILL BE MADE REAL SIMPLE.

CARD NO. ...SYNTACTICAL ERRORS IN ALGOL PROGRAM.

PROGRAM HAS TOO MANY LONG IDENTIFIERS. (OVER 6 CHARACTERS).

THE STRING BEGINNING ON CARD NO. ...HAS NO END QUOTE.

THE TOTAL NUMBER OF CHARACTERS IN ALL LONG IDENTIFIERS IS TOO BIG. SHORTEN SOME IDENTIFIERS.

THE STATE CELLAR HAS OVERFLOWED. TAKE TO COMPILER SECTION.

THE OPEN LOOP ADMISSIBLE VARIABLE LIST HAS OVERFLOWED. TAKE TO COMPILER SECTION.

THE SUBSCRIPT LINEARITY CALCULATION CELLAR HAS OVERFLOWED. TAKE TO COMPILER SECTION.

PROGRAM HAS TOO MANY DIFFERENT CONSTANTS AND STRINGS.

PROGRAM HAS TOO MANY IDENTIFIERS. PLEASE USE CODE PROCEDURES.

PASS 1. CELLAR OVERFLOW.

PASS 1. LONG IDENTIFIER LIST OVERFLOW.

PASS 1. GENERAL IDENTIFIER LIST OVERFLOW.

PBLDS CELLAR OVERFLOW. TAKE TO COMPILER SECTION.

OPERAND CELLAR OVERFLOW. TAKE TO COMPILER SECTION.

PASS 3 ERROR. TAKE TO COMPILER SECTION.

INPUT-OUTPUT PROCEDURE IS USED AS ACT. PAR. (PASS 3 ERROR.)

PROGRAM IS TOO BIG TO FIT IN MEMORY. NO EXECUTION.

MISSING) AT THE END OF THE PROCEDURE CALL.

A...IS ENDED BY A ; BUT IS NOT COMPLETED.

ILLEGAL OCCURRENCE OF CHAR. ...IN OR AFTER...

ILLEGAL ALGOL CHARACTER.

ILLEGAL OCCURRENCE OF CHAR. ...IN OR AFTER...IF AN 'ELSE' FOLLOWS, IT WILL BE ASSUMED TO BELONG TO THIS 'IF'.

ILLEGAL OCCURRENCE OF CHAR. ...IN OR AFTER...ILLEGAL OCCURRENCE OF CHAR. ... AFTER IDENT., NUMBER OR ARRAY ELEMENT. ILLEGAL OCCURRENCE OF CHAR. ...AFTER A STATEMENT OR EXPRESSION.

ILLEGAL OCCURRENCE OF CHAR. ...IN FORMAL PAR. PART.

ILLEGAL OCCURRENCE OF CHAR. ...IN ACTUAL OR FORMAL PAR. POSITION.

ILLEGAL OCCURRENCE OF CHAR. ...AT BEGINNING OF A PROCEDURE BODY.

UNDEFINED DELIMITER.

ILLEGAL OCCURRENCE OF CHAR. ...IN OR AFTER...'THEN' IS MISSING.

MACHINE ERROR. TAKE TO COMPILER SECTION.

TOO MANY 'END'S. A 'BEGIN' HAS BEEN INSERTED BEFORE A CHAR. ...

'IF' CORRESP. TO...IS MISSING.

ILLEGAL OCCURRENCE OF CHAR. ...IT SHOULD ONLY BE IN A SPECIFICATION PART.

ILLEGAL OCCURRENCE OF CHAR. ...COMPILER ASSUMES IT TO BE A).

MISSING) BEFORE CHAR. ...

ILLEGAL OCCURRENCE OF CHAR. ...COMPILER ASSUMES IT TO BE A).

NO CLOSING BRACKET FOR...

ILLEGAL OCCURRENCE OF A STRING.

NO MATCHING FOR LIST FOR THE CHAR. ...

EITHER THE 'ELSE' HAS NO MATCHING 'IF' AND 'THEN' OR THE BLOCK FOLLOWING THE 'THEN' IS NOT YET CLOSED BY 'END'.

ILLEGAL OCCURRENCE OF CHAR. ...IN OR AFTER...IT HAS BEEN REPLACED BY A (.

ILLEGAL OCCURRENCE OF CHAR. ...IN OR AFTER...IT HAS BEEN REPLACED BY A ; .

ILLEGAL OCCURRENCE OF CHAR. ...IN OR AFTER...IT HAS BEEN DROPPED FROM THE SOURCE PROGRAM.

ILLEGAL OCCURRENCE OF CHAR. ...IN OR AFTER...ILLEGAL OCCURRENCE OF CHAR. ... AFTER IDENT., NUMBER OR ARRAY ELEMENT WHICH IS IN OR AFTER A... ILLEGAL OCCURRENCE OF CHAR. ...AFTER A STATEMENT OR EXPRESSION.

ILLEGAL OCCURRENCE OF CHAR. ...ONLY A ; OR 'END' IS EXPECTED.

ILLEGAL OCCURRENCE OF CHAR. ...IN AN IDENTIFIER OR IN A PLACE WHERE ONLY AN ID. WAS EXP.

THE FOLLOWING TOLERATED CHARS. WERE USED IN PROGRAM: ...

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.

9.1. Introduction

This section indicates the form of an object program for an ALGOL program translated by the ALCOR-ILLINOIS 7090/94 Compiler.

It is not intended for beginning programmers, but for those who are experienced in the 7090/94 machine language and have a reasonable knowledge of ALGOL. In order to keep the section to a reasonable size, explanations are not always given, and not always in full detail; the object program produced for any given component of ALGOL should be enough to understand what happens. Also no indication is given as to a proof of why something works - this can be seen from the object program.

This is not a theoretical work, but a section indicating how ALGOL is implemented on a certain machine. Nevertheless, it could be used as a model for any computing machine with somewhat similar instructions.

To help in reading this section, section 9.19 gives a list of all definitions, or a reference to the definitions.

9.2. Conventions

1)	HILOC	is the name of the highest location an object program can use during execution
2)	<B>	contents of location B
3)	$\langle B \rangle_{A(\text{or } P,T,D)}$	contents of location B address (prefix, tag, or decrement)
4)	<A>→B	contents of location A is put into B
5)	>A<	the address of A
6)	$(B)_{A(\text{or } P,T,D)}$	location B address (prefix, tag, or decrement)
7)	AC,MQ,IND	accumulator, multiplier quotient, indicators
8)	XRi	index register i
9)	AC_1	logical AC (p,1...35)
10)	AUXILIARY	general name for a temporary storage location
11)	AUX	general name for a temporary storage location
12)	SIFV	a simple integer variable, which, if a formal parameter, is VALUE
13)	Code procedure	is a procedure whose body does not appear in the program being compiled, but is independent of it. One declares such a procedure as usual, but replaces the procedure body by the delimiter "'CODE'".

9.3. Representation of INTEGER, REAL, BOOLEAN and STRING values

The value of INTEGER declared variables are represented by floating point numbers, as are the values of REAL declared variables.

FALSE and TRUE are represented by

```
OCT  o
```

```
OCT  77777777777
```

respectively.

Strings are written six characters to a word, using ascending locations. After the string the character $(77)_8$ is added. The last word is then filled up with the character $(77)_8$.

9.4. Storage Allocation

9.4.1. Transfer vectors

As in FORTRAN, all names of procedures and functions not appearing directly in this object program (standard functions, I/O procedures, code procedures) are collected and put at the beginning of the object program. The loader then changes these names to TTR subroutine when the program is loaded for execution.

9.4.2. Constants

All constants and constant strings are collected and stored immediately behind the instructions of the object program.

The 5 locations

```
FALSE  OCT  o
TRUE   OCT  777777777777
.ZER   OCT  233oooooooooo
.ONE   OCT  1
.EXTRA PZE
```

appear in every object program as the first 5 locations in the constant list. Location .EXTRA does not contain a constant, but is used as an auxiliary location.

9.4.3. Hierarchy and block numbers

A main program (BEGIN ...) is called a hierarchy or order o. A precompiled procedure (PROCEDURE ... or <TYPE> PROCEDURE ...) is called a hierarchy of order 1. If a procedure (which is not a precompiled procedure) is declared in a hierarchy of order n, it is a hierarchy of order n+1. The hierarchy number of a procedure is then the depth of nesting within other procedures. The term "procedure with hierarchy number k" is sometimes shortened to "hierarchy k". In the same way, the block number of a block in a hierarchy is defined as the level of nesting in other blocks of this hierarchy. An imaginary block o surrounds each hierarchy.

9.4.4. Free fixed storage, FFS, DFS

Each hierarchy k has a set of locations consisting of simple variables, auxiliary locations, array information vectors, hidden variables, etc., whose length can be determined at compile time - called the free fixed storage (ffs) of this hierarchy. The ffs of hierarchy o is given absolute locations at compile time in upper memory, starting at HILOC-1 and working downward (see figure 2). Locations in the ffs of hierarchy $k \neq o$ are given addresses -1, -2, -3, ... which will be relocated with the aid of an index register during run time. Storage for hierarchy $k \neq o$ is therefore allocated dynamically, as is storage for all arrays. In general, parallel blocks will share storage locations in the ffs of the hierarchy in which they appear, since parallel blocks cannot be open at the same time.

When a block in which arrays are declared, or a procedure is entered it takes storage immediately below storage that has been used so far (see figure 2). In order to explain this more clearly, we go into the details of the object program.

Let the locations n to n+m-1 be allocated to ffs of hierarchy k. Then location n+m is known as the FFS_k. FFS_o = HILOC is an absolute location. For $k \neq o$, FFS_k is a variable depending on storage allocation immediately before the procedure was called. When hierarchy k is entered, and FFS_k determined, $-FFS_k$ is put into XR1. Then all locations -i of the ffs of this hierarchy will be referenced by -i,1 (in certain instances by -i,4 if XR4 is properly loaded).

Each block,b, of hierarchy k has a reference location ${}_bDFSL_k$ in the ffs which will contain an address ${}_bDFS_k$ (Dynamic Free Storage) when the block is entered. ${}_bDFS_k$ is the lowest location used by the block for arrays or variables, this is determined by:

${}_oDFS_o$ = lowest location of free fixed storage of hierarchy o.

$FFS_k (k \neq o)$ = DFS of block which called this procedure.

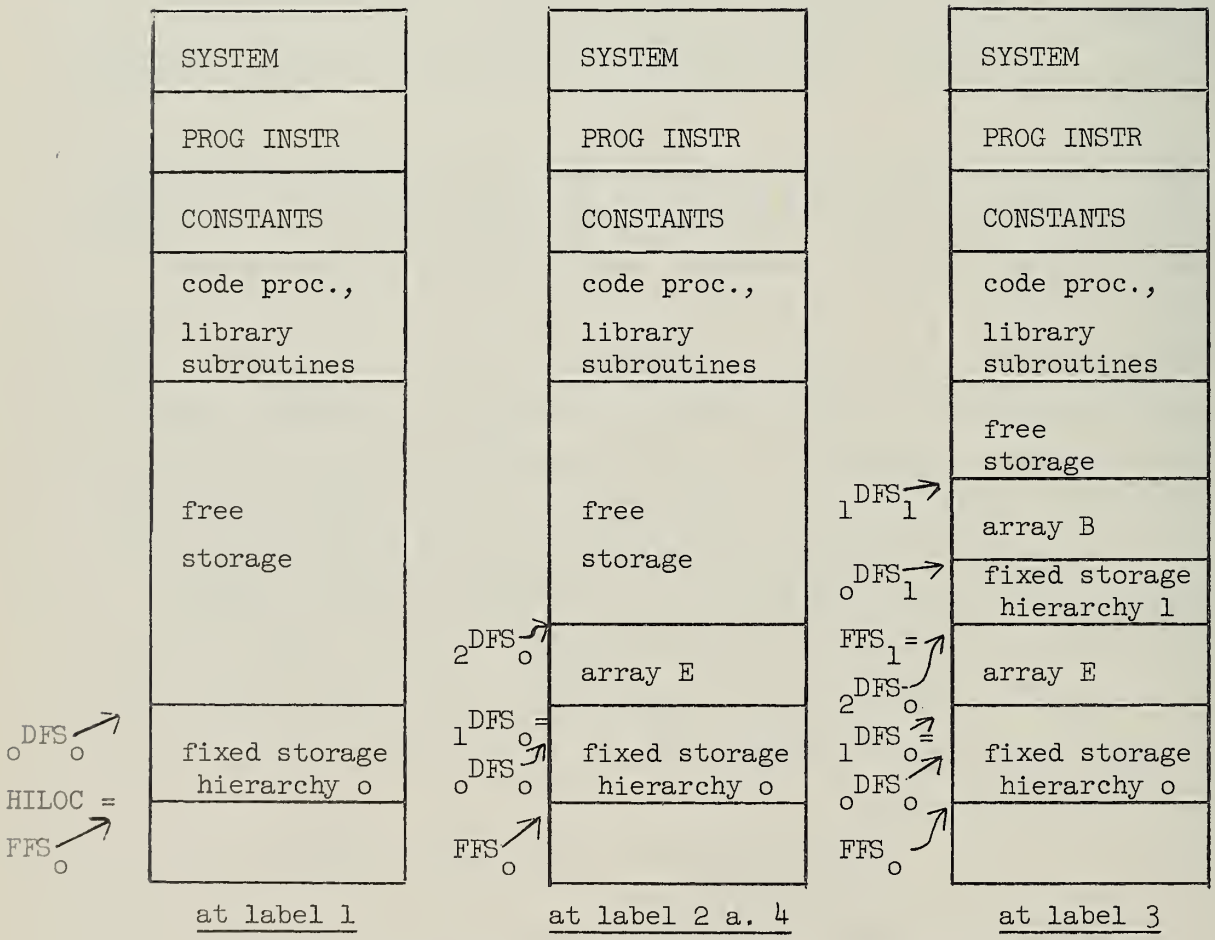
${}_oDFS_k (k \neq o)$ = FFS_k - length (free fixed storage of k) - length (storage needed for value arrays)

${}_bDFS_k (b \neq o)$ = ${}_{b-1}DFS_k$ - length (storage needed for arrays declared in b)

Date:	9/28/64
Section:	9.4.4.
Page:	1 of 2
Change:	1

```
1:  BEGIN PROCEDURE  A;
        BEGIN  ARRAY  B[1:1o];
        3: D.= 5
        END PROCEDURE  A;
        REAL D;
    BEGIN  ARRAY E[1:1o];
    2: A;
    4: END
    END PROGRAM
```

Figure 1



Storage allocation at different points in program in figure 1.

Figure 2

Date:	9/28/64
Section:	9.4.4.
Page:	2 of 2
Change:	1

9.4.5. Object program for storage allocation

Location $_{b}DFSL_{k}$ contains not only the $_{b}DFS_{k}$, but also an instruction:

```
b DFSL k    AXC    b DFS k, 4
```

The object program for storage allocation is as follows:

(a) Beginning of main program:

```
AXC     o DFS o, 4
CAL     * - 1
SLW     o DFSL o
```

(b) when entering block b($\neq$o) of hierarchy k:

```
CAL     b-1 DFSL k
SLW     b DFSL k
```

(c) Array storage allocation is executed by a subroutine which has as a parameter $_{b}DFSL_{k}$, and will change the contents of this location accordingly.

(d) Call of a procedure or formal parameter. If "necessary" the instruction

```
LDI     b DFSL k
```

will appear. This instruction is "necessary" if a label, block begin, block end, or if then else structure appears between this call and the last call of a procedure or formal parameter, or if this is the first call appearing in the program. The indicators are therefore used to convey information about storage allocation, and should not be disturbed by machine language subroutines.

(e) Begin of a procedure (hierarchy k). Procedure linkage is done by a subroutine which will, among other things, perform the following

(1) Save XR1, indicators (FFS, DFS calling)
(2) -<indicators>$_A$ ⟶ XR1 (FFS_k)
(3) -<XR1>- length (free fixed storage hierarchy k)
- length of storage needed for value arrays
→ $({}_oDFSL_k)_A$

(f) End of procedure.

Restore XR1 (FFS of calling hierarchy) and indicators (DFS of calling block).

This frees the storage used by the procedure.

Important

It is to be noted that if $k \neq o$ in (b) and (d), then index register 1 will be used.

That is, if $k \neq o$ then the instruction will appear as

CAL $\quad {}_{b-1}DFSL_k$, 1 instead of

CAL $\quad {}_{b-1}DFSL_k$

In the rest of this section this will not always be mentioned, but it will be assumed that XR1 will be used for all variables in the free fixed storage of hierarchy $k \neq o$ when the program is in this hierarchy k.

9.5. Array Declaration

9.5.1. Information vector and storage of an array

Every array $A[L_1:U_1, \ldots, L_n:U_n]$ has as information vector in the ffs of the form

location A	$U_2 - L_2 + 1$
A+1	$U_3 - L_3 + 1$
	–
	–
	–
A+n-2	$U_n - L_n + 1$
A+n-1	$>A[o,\ldots,o]<$
A+n	DFS_{new}/DFS_{old}

"A", then, refers to the lowest location of the information vector. All numbers are in the address of the word, except DFS_{new}, which is in the decrement.

$$DFS_{old} = >A[U_1,U_2,\ldots,U_n]< +1$$

is the DFS of the block in which the array is declared, before the declaration.

$$DFS_{new} = >A[L_1,\ldots,L_n]< = DFS_{old} - \prod_{i=1}^{n} (U_i - L_i + 1)$$

is the DFS of the block after the declaration. Storage of the array is defined by the rule.

$$>A[i_1,i_2,\ldots,i_n]< = >A[o,o,\ldots,o]< + (\ldots(i_1*(U_2-L_2+1)+i_2)*\ldots) *(U_n-L_n+1)+i_n$$

where $$>A[o,o,\ldots,o]< = DFS_{new}-(\ldots(L_1*(U_2-L_2+1)+L_2)* \ldots)*(U_n-L_n+1)+L_n$$

Date:	9/28/64
Section:	9.5.1
Page:	1 of 1
Change:	1

9.5.2. Subroutine calls in the object program

Consider the declaration in a block with block number b; hierarchy k.

ARRAY $A_n, A_{n-1}, \cdots, A_o [L_1 : U_1, \cdots, L_n : U_n]$

The object program for allocation of storage to A_o is

```
        AXT   o,1   (only if k = o)
        TSX   )ARDEC, 4
        TIX   A_o,,n
        TIX   bDFSLk
(±)     TSX   >L1<
(±)     TSX   >U1<
          -
          -
          -
(±)     TSX   >Ln<
(±)     TSX   >Un<
```

The sign of a TSX is + if the corresponding bound is in floating point, - if in fixed point. The bound will be in fixed point if it is an unsigned integer. $>L_i<$ (or $>U_i<$) is an address or an address with tag 1 (indicating FFS_k). If a bound is not addressable in this form, it is evaluated immediately before the call of)ARDEC and put in an auxiliary location in ffs and this auxiliary location is used.

For $i \neq o$, storage is allocated to A_{i-1} immediately before storage is allocated to A_i. The object program for A_1 is

```
TSX   )ARDEI, 4
TIX   A_i-1,,n
TIX   bDFSLk,,Ai
```

Date:	9/28/64
Section:	9.5.2.
Page:	1 of 1
Change:	1

9.5.3. Subroutines)ARDEC and)ARDEI

Subroutine)ARDEC does the following:

a) make up the information vector for A_o

b) check for overflow of memory

c) change contents of ${}_bDFSL_k$

Subroutine)ARDEI does the following:

a) copy the information vector of A_{i-1} into locs A_i, A_i+1, ..., A_i+n

b) check for overflow of memory

c) reduce (in inf vect for A_i) the address part of loc A_i+n, A_i+n-1, and the decrement of A_i+n by the length of the array

d) change location ${}_bDFSL_k$

Date:	9/28/64
Section:	9.5.3.
Page:	1 of 1
Change:	1

9.6. Strings

Strings are also referenced by an information vector of the form

PZE loc of first word of string,, number of words

(one word only).

9.7. Switch Declaration

SWITCH S := D_1, D_2, ... , D_n

where D_i is a designational expression.

The object program is:

```
        TRA   OVER
G1:     calculate D1 and jump
G2:     calculate D2 and jump
                -
                -
                -
Gn:     calculate Dn and jump
S       EQU   *-1
        TRA   G1
        TRA   G2
                -
                -
                -
        TRA   Gn
OVER    BSS   o
```

See section 9.14. for the instructions "calculate D_i and jump".

As can be seen, S is given the address >(TRA G_1)< -1

If the set of instructions for

```
Gi:  calculate Di and jump
```

consists of a single instruction (in most cases), then the set of instructions at G_i is deleted, and TRA G_i is replaced by calculate D_i and jump.

The jump to a switch element, S[E], is then accomplished by putting -E into XR4 and executing a

```
TRA   S, 4
```

Date:	9/28/64
Section:	9.7.
Page:	1 of 1
Change:	1

9.8. Procedure Declaration

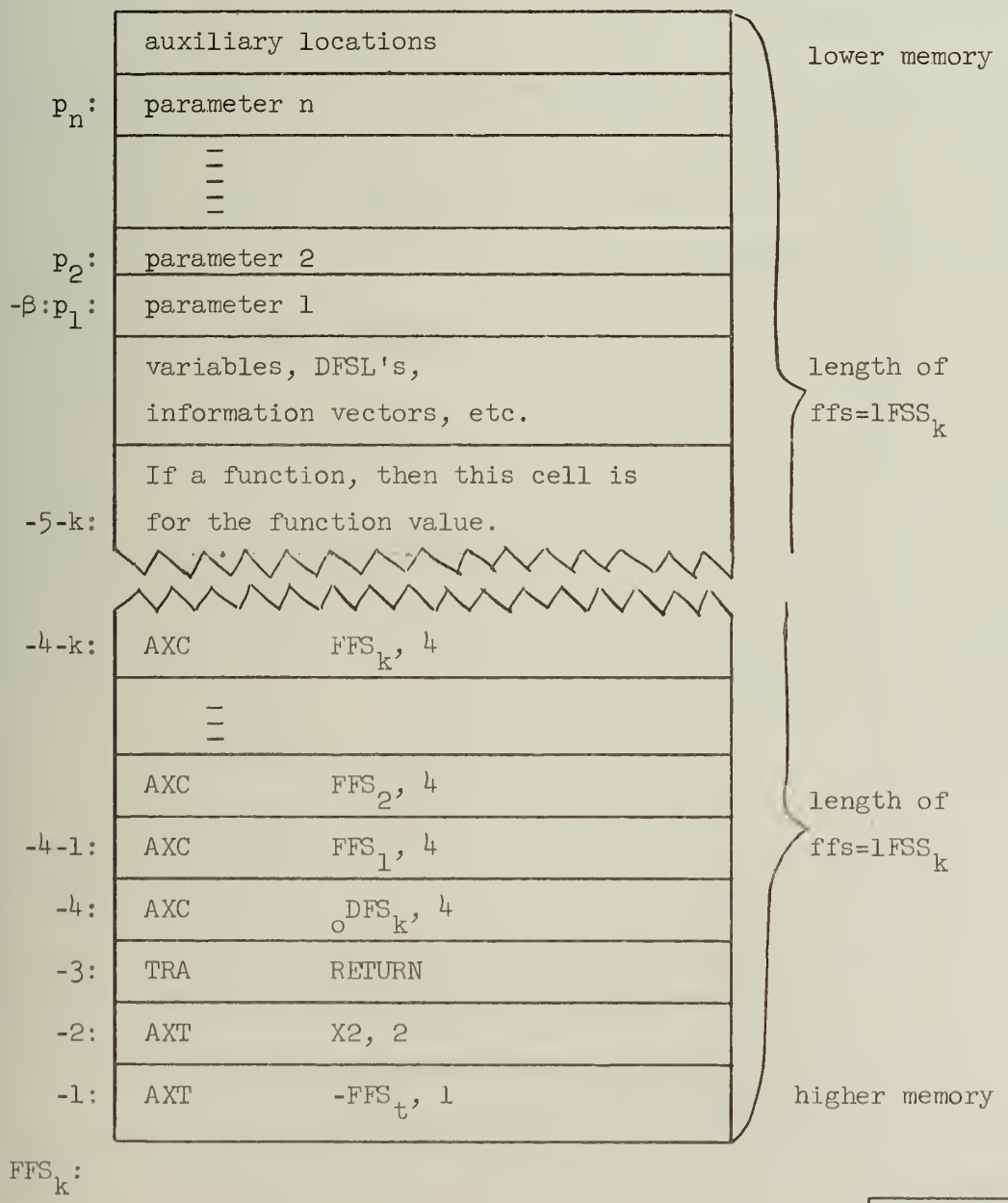
Naturally, the procedure declaration is closely connected to the procedure call and the call of formal parameters. It would be best, then, to glance over sections 9.8., 9.10. and 9.11., and then to study all three together in detail. Important for all three sections is the layout of the ffs of a procedure when in the procedure, given in 9.8.1.

Let k be the hierarchy number of the procedure being declared, b the block number of the block and t the hierarchy number in which the call of this procedure occurs. X2 is the contents of XR2 at the call and RETURN is the return address. Let this procedure have n parameters.
RETURN = calling point +n+1.

9.8.1. ffs of a procedure

At execution time, when in a procedure declaration of hierarchy k there is always exactly k-1 surrounding procedure declarations. These procedures are open, and have hierarchy numbers k-1, k-2,..,1, and corresponding FFS's FSS_{k-1}, ... ,FSS_1. Each formal parameter uses 1 location in the ffs, except an array of dimension n, which takes n+1 locations.

The following is the layout of the ffs.



p_i denotes the lowest location used for the parameter i, -β denotes the highest location used by the first parameter. Each ffs_k, $k \neq o$, has at least 5 auxiliary locations, which are used by the procedure transporter routine, A)PTRA.

Upon entering the procedure, if formal parameter i is ARRAY, the information vector is transported into ffs_k. If it is also VALUE, the array is then transported to locations just below this $_oDFS_k$, the information vector which has been transported to this ffs is changed accordingly, and location -4 of FFS_k (the $_oDFSL_k$) is changed.

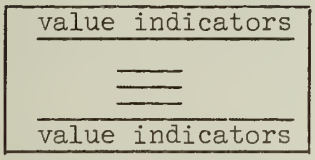
If formal parameter i is not ARRAY and is called by VALUE, the value will be stored in the location p_i. Otherwise an instruction will be stored in the location p_i. (See procedure call and formal parameter call, sections 9.10.6. and 9.11., and section 9.8.3.).

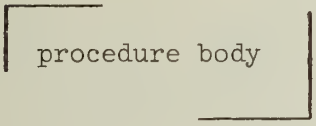
9.8.2. Object program for procedure declaration

```
                TRA   OVER
```

S,1...8 9 13 14 20 21 35

```
                PZE   -β
                PZE   -1FFS_k
entry point:    PXA   o,4
                TSX   A)PTRA, 4
```






```
                CLA or CAL  -5-k    (if this is a function)
                XEC         -2,1    (restore XR2)
                LDI         -4-k,1  (restore indicators)
                LDQ         -3,1    (initialize RETURN)
                STQ         *+2
                XEC         -1,1    (restore XR1)
                PZE                 (return)
     OVER       BSS         o
```

Each word of "value indicators" has the following form:

S1- 9 10 11 12 13 32 33 34 35

Date:	9/28/64
Section:	9.8.2.
Page:	1 of 2
Change:	1

A o (1) in an even bit means that this parameter is (not) VALUE.

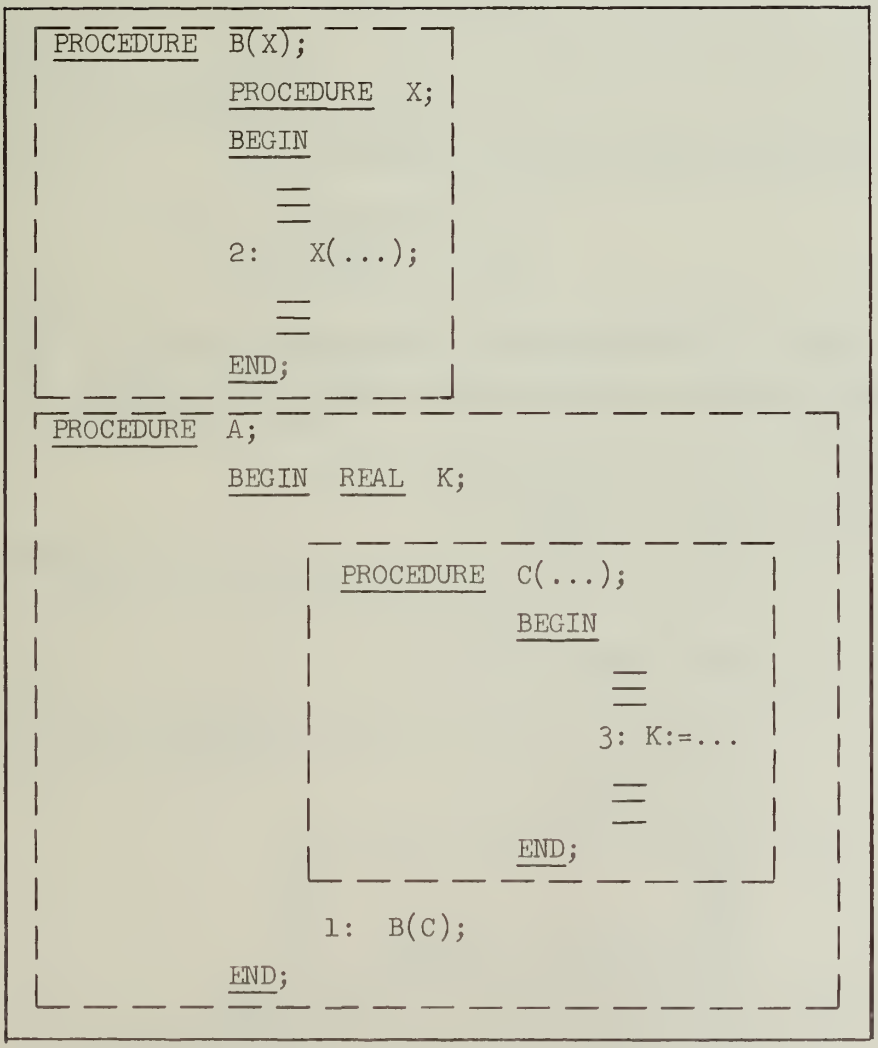
As many words as necessary are used to indicate VALUE or not VALUE for all parameters.

9.8.3. Subroutine A)PTRA

This subroutine, called at the beginning of every ALGOL procedure, does all necessary storage allocation for the procedure and transports all information about the actual parameters. The parameters for A)PTRA are the value indicators and the 3 locations appearing before the call point (see 9.8.2.).

Some complications occur if the procedure called is a formal parameter.

Consider the following part of a program:



Procedure C, the nested procedure, uses a global value, K. Therefore it must be able to reference ffs_A the free fixed storage of procedure A, since K is declared there. Now C is called, through the use of the procedure formal parameter, at label 2. Procedure C will have to save FFS_B and initialize XR1 upon return with FFS_B. In this case, where a procedure formal parameter is used, two FFS's must be used by the called procedure. We name the first point of call of a formal parameter procedure, in this case at label 2, the official calling point. The pseudo calling point is the point where the actual parameter which is not a formal parameter appears - in this case at label 1.

In most cases, official and pseudo calling points are the same; only with procedures which are formal parameters are they different. FFS pseudo calling point is needed only to get the FFS's of all surrounding procedures. Just how these two are used can be seen from the flow chart of A)PTRA.

In the beginning of the main program appears the instruction

```
STZ    HILOC
```

If <HILOC> is not o, $<HILOC>_D$ contain $-FFS_{\text{official calling pt.}}$ (put in at the official calling point), and XR1 contains $-FFS_{\text{pseudo calling pt.}}$. If <HILOC> = o, the two FFS's are the same and appear in XR1.

The flow chart for A)PTRA is:

a) Allocate storage for ffs, using the DFS appearing in the indicators. (Use the formula given in 9.4.4.)

b) Check memory overflow.

c) Make up words -4-1 to -4-k+1 of the ffs. These are taken from the same part of the ffs defined by XR1 (pseudo calling point).

d) <HILOC> = o? ——— yes →
   no ↓
   $<HILOC>_D$ →XR1, o→ HILOC
   ↓ ←

e) Make up word -1 to -4 of the FFS.
   Make up word -4-k (this is the indicators).

f) Move the parameter list as follows (refer to section 9.10.6.3.).

1) Parameter i is array or string (dimension n).

a) Execute thunk th_i

b) Move information vector from locations defined by <XR4> through <XR4> + n to locations p_i through p_i+n.

c) If value array, move the array to free storage directly below location $_{o}DFS_k$, change $_{o}DFSL_k$, check for memory overflow, change the information vector.

2) Parameter is value.

executive thunk th_i, put value in location p_i

3) Parameter not value.

$<t_i> \longrightarrow p_i$

g) Fix the indicators with $_{o}DFS_k$ and XR1 with $-FFS_k$.

h) Return to the cell directly after the value indicators.

9.9. Globals

Suppose the program is in hierarchy k, and a name X, which is declared or specified in hierarchy t, t<k is used. X is then a "global variable".

9.9.1. Declared in hierarchy o

If t = o, nothing extra is required, since X has an absolute location.

9.9.2. Declared or specified in hierarchy $\neq$ o

$t \neq o$. XR1 contains $(-FFS_k)$ and not $(-FFS_t)$. Therefore something extra is required. The instruction

```
XEC     -4-t,1
```

is executed, which puts into XR4 the value $(-FFS_t)$. (If XR4 already contains $(-FFS_t)$, this instruction will not appear in the object program.) X is then referred to using XR4 instead of XR1. (See section 9.8.1. - ffs allocation.)

In what follows, it will be taken for granted that this will be done, without mentioning it.

9.10. Procedure Call.

A function returns its value in the AC.

9.10.1. SIN, COS, SQRT, LN, EXP, ARCT

These have an object program

```
argument → AC
TSX   subroutine, 4
```

or

```
TSX   subroutine, 4
TSX   >argument<
```

depending on the installation. It is also a parameter of the compiler whether >argument< should be allowed to have an index register or not.

9.10.2. ABS, SIGN, ENTIER

These three are open subroutines.

ABS

```
argument → AC
SSP
```

SIGN

```
argument → AC
TZE  *+3
CLM
ORA  >1.o<
```

ENTIER

```
argument → AC
TPL  *+6
TZE  *+7
CAS  >-1.o<
CLA  >-1.o<
TRA  *+4
FSB  >.999...999<
UFA  .ZER
FAD  .ZER
```

9.10.3. a↑b

If b is not one of the unsigned integers 1,2,3, or 4, this is done through a subroutine (one for integer b and one for real b). The object program is

```
a → AC
b → MQ
TSX    subroutine, 4
```

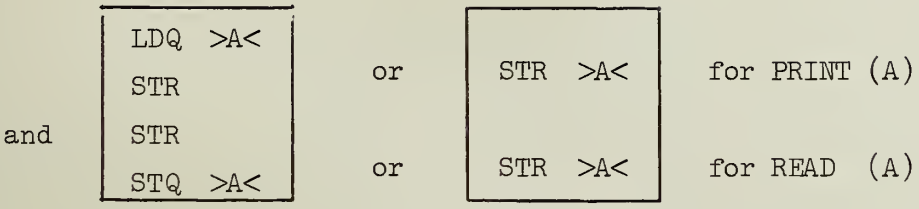
or

```
TSX    subroutine, 4
TSX    >a<
TSX    >b<
```

depending on the installation. See section 9.13.2.

9.10.4. PRINT, PRINTF, READ, READF

These have the form of FORTRAN I/O routines. There is a choice between the 2 methods



See a FORTRAN manual for details. The format location and the tape number are not in the calling sequence at the time of the call, but are put in by the called subroutine, (which is connected to the FORMAT subroutine) which then jumps into the installation's I/O package.

9.10.5. READMATRIXF, etc.

These have the same format as given in 9.10.4. The object program for I/O of the array itself, assuming PRINTMATRIX(A), is:

```
<A + n> --> AC
STA         *+4
PAX         0,4
STD         *+1
TIX         *+1,4,**
STR         **,4
TIX         *-1,4,1
```

where A + n is the last word of the information vector.

9.10.6. Usual procedure call

Let the call take place in a block with block number b and hierarchy k. The procedure has n parameters.

9.10.6.1. Thunks

A thunk is the set of coding which evaluates an actual parameter. Usually a thunk is this compiler will deliver, in XR4, the complement of the address where the value of the actual parameter is. Exceptions are procedures and arrays as actual parameter. A thunk has the following form:

k = o

```
body of thunk
TRA   2,2
```

k ≠ o

```
PXA   o,1      save FFS calling
XEC*  1,2      put -FFS_k of thunk
               in XR1
STO   AUX,1    save FFS calling
body of thunk
CLA   AUX,1    restore
PAX   o,1      FFS calling
TRA   2,2
```

An exception is the case of a procedure, or function with parameters as actual parameter, and k ≠ o. The second instruction is

`XEC*  -1,2` instead of `XEC*  1,2`

In order to understand the beginning and end of a thunk, it is necessary to be familiar with the call of a formal parameter (section 9.11. and section 9.10.6.3.).

Date:	9/28/64
Section:	9.10.6.1.
Page:	1 of 1
Change:	1

9.10.6.2. Body of a thunk

The following is the object program for the body of a thunk if the actual parameter is a:

a) Constant, simple variable declared in hierarchy t, or value formal parameter specified in hierarchy t. The thunk delivers (-address of variable) into XR4.
t = o, or actual parameter a constant

```
AXC     address, 4
```

o<t<k

```
XEC     -4-t, 1
SXD     *+2,4
AXC     address, 4
TXI     *+1,4,**
```

t = k

```
SXD     *+2,1
AXC     address, 4
TXI     *+1,4,**
```

b) Simple variable or function without parameters which is a formal parameter

```
object program the same as the
call of formal parameter (section 9.11.4.)
```

c) Array name, A, or string, declared or specified in hierarchy t. The thunk delivers the address of the first imformation vector location in XR4

t = o

```
AXT     A, 4
```

t = k

```
PXA     o,1
PAC     o,4
TXI     *+1,4,A
```

o<t<k

```
XEC     -4-t,1
PXA     o,4
PAC     0,4
TXI     *+1,4,A
```

d) Arithmetic or boolean expression.

The thunk delivers ->value of expression< into XR4.

```
value of exxpression—>AC
STO   (SLW)   .EXTRA
AXC           .EXTRA,4
```

e) Designational expression (see jumps-section 9.14.)

```
calculate expression and jump
```

f) Switch name, S, not formal parameter.

It is expected that XR4 contains (-switch element number to jump to). (See section 9.14.)

```
TRA   S, 4
```

g) Procedure (or function with par), not formal parameter, not standard

```
LAC   HILOC, 4
TXI   Procedure, 4, -2
```

-(official calling pt.)—> XR4

h) Procedure (or function with par), formal parameter (the i^{th} parameter)

```
NOP   -1,1
XEC   Pi, 1
```

i) Standard procedures or functions.

Standard I/O procedures are not allowed as actual parameters.

Suppose SIN is the actual parameter. Since the calling sequences are different for standard functions and usual functions, in order to assure full recursiveness, the program must be treated as if the actual

parameter is another function name, say SIN1, and that SIN1 is declared as:

```
REAL PROCEDURE  SIN1 (X);
                VALUE X;  REAL X;
                SIN1 .= SIN(X);
```

The body of the thunk is then

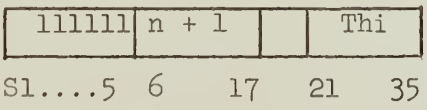
```
LAC   HILOC,4          get -(official calling
TXI   *+4,4,-2         point    ) into XR4
PZE   o,1
PZE   -8               See procedure
PZE   -13
PXA   o,4              declaration-section 9.8.2.
TSX   A)PTRA,4
OCT   oool77777776
CLA   -8,1
TSX   SIN, 4
XEC   -2,1
LDI   -5,1
LDQ   -3,1
STQ   *+2
XEC   -1,1
PZE
```

9.10.6.3. Procedure call object program

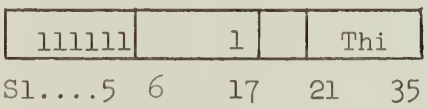
If the procedure is a formal parameter, the instruction at location OVER is replaced by others, as shown in section 9.11.2.

```
        LDI    bDFSLk    (only if not already loaded,
                          or not in a thunk)
        TRA    OVER
  Th1   thunk for parameter 1
  =      =
  =      =
  Thn   thunk for parameter n
 OVER   TSX   PROCEDURE, 4
  t1   ┌ one location for
  =    =  each parameter
  =    =
  tn   ┘
```

a) If the i^{th} parameter is an array of dimension n, location ti is



b) If the i^{th} parameter is a string, location ti is



c) If the body of the thunk i (ti not an array or string) is a single instruction (except a jump to hierarchy $\neq$ o), then Thi is deleted and ti is the single instruction.

d)

ti is `TSX  Thi,2`

Now, to get the value of the actual parameter i, one has in essence only to execute the instruction ti, or in the case of array or strings, execute an instruction

`TSX*  ti,2`

See formal parameter call (section 9.11.).

Date:	9/28/64
Section:	9.10.6.3.
Page:	1 of 1
Change:	1

9.11. Call of Formal Parameter

Suppose the call occurs in hierarchy k, the parameter itself belongs to hierarchy t.

9.11.1. Arrays, strings

An array or string, whether VALUE or name, is treated as declared in the procedure of which it is a formal par., since the information vector has been transported by subroutine A)PTRA into the ffs.

9.11.2. Procedure, or function with parameter

t = k

```
STL     HILOC
SXD     HILOC,1
NOP     -1,1
XEC     pi,1
```

t < k

```
XEC     -4-t,1
STL     HILOC
SXD     HILOC,1
NOP     -1,4
XEC     pi,1
```

The STL instruction saves the (official calling point -2) of the procedure. Next the FFS of the official calling point is saved. The NOP instruction is a reference to the FFS of the hierarchy to which the actual parameter belongs. The rest of the procedure call is exactly as described in section 9.10.6.3. These instructions replace the instruction OVER TSX procedure, 4 of section 9.10.6.3.

9.11.3. All others, VALUE

The value appears in location pi of FFS_t, and is referenced as any variable declared in hierarchy t.

9.11.4. All others, name

t = k

```
XEC     pi, 1
NOP     -1,1
```

t < k

```
XEC     -4-t,1
XEC     pi, 4
NOP     -1,4
```

As in 9.11.2., the XEC instruction causes the execution of thi. In this case XR4 contains (-address of location) upon return.

If this formal parameter call occurs in a thunk, the instructions

```
PXA     0,2
STO     AUX,1
```

```
CLA     AUX,1
PAX     0,2
```

appear before and after the call respectively.

9.12. Array Element Address Calculation.

The object program uses the information vector and the allocation rule given in section 9.5.1.

An expression in a subscript position will be in fixed point if it is an unsigned integer. In this case the instructions for unfloating are deleted. The instructions for UNFLOAT are

```
UFA     .ZERO
LGL     lo
LGR     lo
```

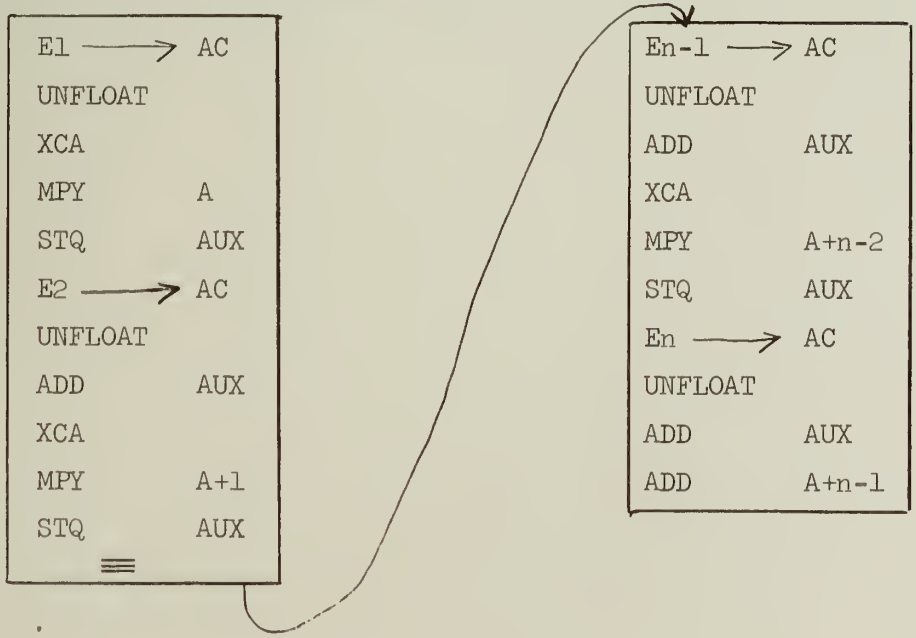
The array element address calculation leaves the address in the AC. It will then be put in XR4 (complemented) if it can be used immediately, or stored in an auxiliary location - to be used indirectly.

Consider the array element A [E1], where E1 is any expression. The object program is:

```
E1 →AC
UNFLOAT
ADD   A
```

where A is, as usual, the first (lowest) location of the information vector.

Consider the array element A[E1,E2,...,En]. The object program is:



9.13. Operations and Relations

Overflow and underflow are handled by the installation's floating point trap routine (it is assumed the machine is in floating point trap mode). Therefore this depends on the system. The compiler can be altered easily through the use of parameters to insert nothing at the beginning of a main program or the instructions

```
CLA     (FPT)
STO     8
```

or

```
TSX     (FPT), 4
```

where (FPT) is the name of a subroutine and is in the transfer vector, in order to initialize the floating point trap routine.

The instruction FDP is used for division. No check is made by the compiler for a divide check.

The result is always left in the AC for arithmetic, AC_1 for boolean operations or relations. The instruction CLA A or CAL A stands for "calculate the expression A and put it in the AC". If there are two alternatives for a certain operation, the second one indicates the object program if an operand is already in the AC. If there is only one, it is assumed that if the first operand is already in the AC, the CLA (or CAL) is deleted.

The operands of commutative operations are exchanged if the second operand is already in the AC.

9.13.1. NOT and unary minus.

NOT A

```
CAL     A
COM
ANA     TRUE
```

-A

```
CLS     A
```

or

```
CHS
```

9.13.2. Arithmetic operations

A + B

```
CLA   A
FAD   B
```

A - B

```
CLA   A
FSB   B
```

or

```
CHS
FAD   A
```

A / B

```
CLA   A
FDP   B
XCA
```

A * B

```
LDQ   A
FMP   B
```

or

```
XCA
FMP   B
```

A // B

```
CLA   A
FDP   B
XCA
UFA   .ZER
FAD   .ZER
```

A ↑ B

If B is not one of the 4 unsigned integers 1,2,3 or 4, this is done by a subroutine (see section 9.10.5.)

B = 1

```
CLA   A
```

B = 2

```
LDQ   A
FMP   A
```

OR

```
STO   AUX
XCA
FMP   AUX
```

B = 3

```
LDQ   A
FMP   A
XCA
FMP   A
```

or

```
STO   AUX
XCA
FMP   AUX
XCA
FMP   AUX
```

B = 4

```
LDQ     A
FMP     A
STO     AUX
XCA
FMP     AUX
```

or

```
STO     AUX
XCA
FMP     AUX
STO     AUX
XCA
FMP     AUX
```

9.13.3. Boolean operations.

A AND B

```
CLA   A
ANA   B
```

A OR B

```
CLA   A
ORA   B
```

A IMPLIES B

```
CAL   A
COM
LBT
CAL   B
```

A EQUIV B

```
CAL   A
ERA   B
COM
ANA   TRUE
```

9.13.4. Relations.

The relation yields a boolean value. In case a relation occurs as the boolean expression between IF and THEN, the object program is more efficient. See section 9.15.4.

A EQUAL B

```
CLA   A
FSB   B
TZE   *+2
CAL   TRUE
COM
ANA   TRUE
```

A NOTEQUAL B

```
CLA   A
FSB   B
TZE   *+2
CAL   TRUE
```

A LESS B

```
CLA   A
FSB   B
TZE   *+4
TPL   *+3
CAL   TRUE
LBT
PXD   0,0
```

A GREATER B

```
CLA   A
FSB   B
TZE   *+4
TMI   *+3
CAL   TRUE
LBT
PXD   0,0
```

A NOTLESS B

```
CLA   A
FSB   B
TZE   *+2
TMI   *+3
CAL   TRUE
LBT
PXD   0,0
```

A NOTGREATER B

```
CLA   A
FSB   B
TZE   *+2
TPL   *+3
CAL   TRUE
LBT
PXD   0,0
```

9.13.5. :=

a) A, B boolean

```
CAL   B
SLW   A
```

b) A real, or A,B integer

```
CLA   B
STO   A
```

c) A integer, B real

```
CLA   B
UFA   .ZER
FRN
FAD   .ZER
STO   A
```

9.14. Jumps

The designational expression

GOTO IF B THEN D1 ELSE D2 has the same object program as

IF B THEN GOTO D1 ELSE GOTO D2.

9.14.1. In the same hierarchy or to hierarchy o. (GOTO D).

Jumps in the same hierarchy or to hierarchy o will be simple TRA's since, in the first case FFS stays the same, and in the second it is absolute (XR1 not needed).

9.14.2. To another hierarchy t ≠ o. (GOTO D)

The FFS changes. The object program is:

```
CLA    -4-t, 1
PAC    o,1
TRA    D
```

$-FFS_t \rightarrow$ XR1

9.14.3. To a formal parameter.

To a simple label, the call is exactly the same as the call of a simple variable formal parameter (section 9.11.4.). To a switch element S [I] where S is a formal parameter:

```
I ---> AC
UNFLOAT
PAC     o,4
call formal parameter
```

If S is a global (of hierarchy t), the call is

```
-(I) -> XR4
CAL     -4-t,1
PAC     o,1
XEC     S,1
NOP     -1,1
```

9.14.4. To a switch element (GOTO S [E1])

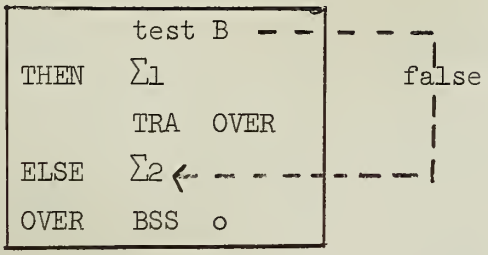
```
E1          AC
UNFLOAT
PAC         o,4
TRA         S, 4
```

(See section 9.7.)

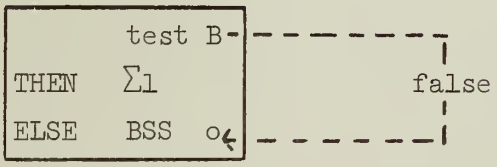
9.15. Conditional Statements and Expressions.

9.15.1. Form of conditional statement.

(a) IF B THEN $\Sigma 1$ ELSE $\Sigma 2$



(b) IF B THEN $\Sigma 1$

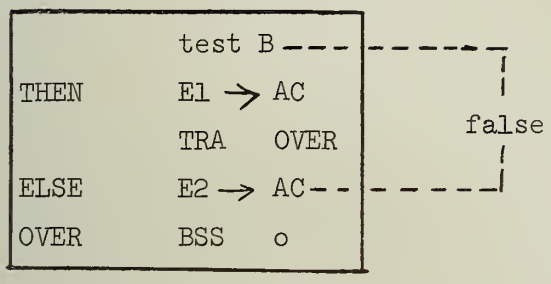


Date:	9/28/64
Section:	9.15.1.
Page:	1 of 1
Change:	1

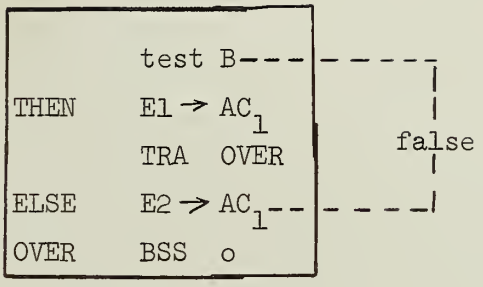
9.15.2. Form of arithmetic and boolean conditional expression.

IF B THEN E1 ELSE E2

arithmetic



boolean



If one of E1 or E2 are real, the type of the expression is real.

Date:	9/28/64
Section:	9.15.2.
Page:	1 of 1
Change:	1

9.15.3. Form of designational conditional expression.

The form of IF B THEN D1 ELSE D2 is the same as that of
IF B THEN GOTO D1 ELSE GOTO D2.

9.15.4. Form of the boolean expression test.

(a) If the boolean expression has the form:
A rel b, where rel is an arithmetic relation,
the object program is:

A EQUAL B

```
CLA   A
FSB   B
TNZ   ELSE
```

A NOT EQUAL B

```
CLA   A
FSB   B
TZE   ELSE
```

A LESS B

```
CLA   A
FSB   B
TPL   ELSE
TZE   ELSE
```

A GREATER B

```
CLA   A
FSB   B
TMI   ELSE
TZE   ELSE
```

A NOT LESS B

```
CLA   A
FSB   B
TZE   *+2
TMI   ELSE
```

A NOT GREATER B

```
CLA   A
FSB   B
TZE   *+2
TPL   ELSE
```

(b) For general boolean expressions the object program is:

```
B ⟶ AC1
TZE     ELSE
```

Date:	9/28/64
Section:	9.15.4.
Page:	1 of 1
Change:	1

9.16. FOR Loops.

Sections 9.16.1. to 9.16.4. give definitions which are necessary to differentiate different types of for loops and to help define those array elements appearing in a for loop whose address can be calculated by "linear address calculation" (section 9.17.).

The term "is changed" means that the variable occurs on the left hand side of an assignment statement or is a parameter of an input procedure.

9.16.1. Type of FOR list elements.

(a) E1

(b) E1 WHILE B1

(c) E1 STEP E2 UNTIL E3

A FOR loop, then, looks as follows:

FOR I.= FL1, FL2,..., FLn DO

Σ;

where the FLi are FOR list elements, I is the loop variable, and Σ is the FOR statement.

9.16.2. Admissible and proper admissible variables.

The admissible variables of a FOR loop are the

(1) SIFV occuring in E1, E2, or E3 of the FOR list elements,

(2) The loop variable, if it is an SIFV, and

(3) admissible variables of surrounding FOR loops.

An admissible variable is called proper if it is the loop variable of this FOR loop, or if it occurs in "E2" of a FOR list element of this FOR loop.

9.16.3. Types of FOR loops.

1. A FOR loop is called <u>almost feasible</u> if it satisfies the following conditions:

   (a) It has only one FOR list element - and that is of type (c). I.e. - the loop is:
   <u>FOR</u> I.= E1 <u>STEP</u> E2 <u>UNTIL</u> E3 <u>DO</u> Σ;

   (b) The loop variable is an SIFV.

   (c) E2 consists of SIFV's and/or integer constants.

   (d) E1,E2,E3 are of type integer, and contain only the operations +, -, and *.

   (e) Σ contains no procedure call which is not a standard procedure (I/O procedures, etc.).

   (f) If this FOR loop is not in hierarchy o, then Σ and E3 contain no function call, other than a call of a standard function.

   (g) Σ contains no array declaration.

   (h) E1,E3, and Σ contain no call of a formal parameter simple variable which is not value.

   (i) Any admissible variable which is changed in Σ is not a proper admissible variable of this loop.

2. A loop is <u>unfeasible</u> if it is not almost feasible.

3. A loop is feasible if it is almost feasible and no admissible variable of the loop is changed.

4. A loop is <u>perfect</u> if it is feasible and

   (a) E3 consists only of admissible variables and constants.

   (b) E2 is an integer constant.

   (c) Any perfect loop contained in Σ does not contain a jump out of itself, or a designational expression other than a simple label.

Most loops are either perfect or feasible.

9.16.4. Object program for unfeasible loops.

(1) FOR I.= E1, or, E1

hierarchy o

```
E1 ⟶ I
AXT   *+3,4
SXA   Σend, 4
TRA   STATE.
```

hierarchy ≠ o

```
E1 ⟶ I
AXT   *+4,4
PXA   o,4
STO   AUX1, 1
TRA   STATE.
```

STATE. is the first instruction of Σ, the loop statement.

Σ_{end} - see below.

(2) FOR I.= E1 DO ; or , E1 DO ;

hierarchy o

```
      E1 ⟶ I
      AXT   Σend+1,4
      SXA   Σend
STATE. Σ
      TRA   **
Σend
```

hierarchy ≠ o

```
      E1 ⟶ I
      AXT   Σend+1,4
      PXA   o,4
      STO   AUX1, 1
STATE. Σ
Σend  TRA*  AUX1, 1
```

(3) FOR I.= E1 WHILE B1, or , E1 WHILE B1,

hierarchy o

```
T     E1 ⟶ I
      AXT   T,4
      SXA   Σend, 4
      B1 ⟶ AC
      TNZ   STATE.
```

hierarchy ≠ o

```
T     E1 ⟶ I
      AXT   T,4
      PXA   o,4
      STO   AUX1, 1
      B1 ⟶ AC
      TNZ   STATE.
```

(4) FOR I.= E1 WHILE B1 DO or , E1 WHILE B1 DO

Hierarchy o

```
        T     E1 → I
              AXT   T,4
              SXA   Σend, 4
              B1 → AC
              TZE   Σend+1
STATE.        Σ
              TRA   **
Σend
```

Hierarchy ≠ o

```
        T     E1 → I
              AXT   T,4
              PXA   o,4
              STO   AUX1,1
              B1 → AC
              TZE   Σend+1
STATE.        Σ
              TRA*  AUX1,1
Σend
```

(5) E1 STEP E2 UNTIL E3 (DO or ,)

Hierarchy o

```
              E1 → I
              AXT   F3+1,4
              SXA   F3,4
F1            E2 → AC
              STO   AUX2
F3            TRA   **
              AXT   F1,4
              SXA   Σend,4
              AXT   *+3,4
              SXA   F3,4
              PXD   o,o
              FAD   I
              STO   I
              FSB   E3
```

DO	
TZE *+5	TZE STATE.
LDQ AUX2	LDQ AUX2
TQP *+2	TQP *+2
CHS	CHS
TPL Σend+1	TMI STATE.
STATE. Σ	
Σend TRA **	

Hierarchy ≠ o

```
              E1 → I
              AXT   F3+1,4
              PXA   o,4
              STO   AUX3,1
F1            E2 → AC
              STO   AUX2,1
F3            TRA*  AUX3,1
              AXT   F1,4
              PXA   o,4
              STO   AUX1,1
              AXT   *+4,4
              PXA   o,4
              STO   AUX3,1
              PXD   o,o
              FAD   I
              STO   I
              FSB   E3
```

DO	
TZE *+5	TZE STATE.
LDQ AUX2,1	LDQ AUX2,1
TQP *+2	TQP *+2
CHS	CHS
TPL Σend+1	TMI STATE.
STATE. Σ	
Σend TRA* AUX1,1	

9.16.5. Object program for almost feasible and feasible loops.

This should be read in connection with section 9.17.

E2 is an unsigned integer

```
        E1 → I
        TRA   FLIN
F1      CLA   I
        FAD   E2
        STO   I
F2      FSB   E3
        TZE   *+2
        TPL   FEND
STATE.  Σ
        (increment array element addresses (section 9.17.6.))
        TRA   F1
        (subroutines to calculate array element addresses (section 9.17.4.))
FLIN    (initialize array element addresses (section 9.17.5.))
        I → AC
        TRA   F2
FEND    BSS   o
```

E2 not an unsigned integer

```
        E1 → I
        E2 → AUX2
        TRA   FLIN
F1      CLA   I
        FAD   AUX2
        STO   I
F2      FSB   E3
        TZE   *+5
        LDQ   AUX2
        TQP   *+2
        CHS
        TPL   FEND
STATE.  Σ
        (increment array element addresses (section 9.17.6.))
        TRA   F1
        (subroutines to calculate array element addresses (section 9.17.4.))
FLIN    (initialize array element addresses section 9.17.5.)
        I → AC
        TRA   F2
FEND    BSS   o
```

Date:	9/28/64
Section:	9.16.5.
Page:	1 of 1
Change:	1

9.16.6. Object program for perfect loops.

This should be read in connection with section 9.17.

```
                E1 ⟶ I
                E3 ⟶ AC
                FSB    I
                UFA    CONSTLOC
                STO    AUX_S
                UFA    E2
                TMI    F2
                PAX    0,2
                TRA    FINIT
STATE.          Σ
                CLA    I
                FAD    E2
                STO    I

              ( increment array    )
              { element addresses  }
              ( (section 9.17.6.)  )

                TIX    STATE.  ,2,E2
F2              TRA    FEND

              ( subroutines to calculate  )
              { array element addresses   }
              ( (section 9.17.5.)         )

FINIT         ( initialize array    )
              { element addresses   }
              ( (section 9.17.5.)   )

                TIX    STATE.  ,2,E2
FEND            BSS    0
```

CONSTLOC, made up at compile time, contains the value 1 in the address part, with a characteristic of 233. If E1 is an unsigned integer, as is usually the case, the instruction FSB I is omitted and CONSTLOC contains the value -E1+1. Location AUX_S is used in the initializing of array element addresses.

If this perfect loop is imbedded in another perfect loop, an instruction

```
SXA    FEND, 2
```

is inserted after E1 ⟶ I , and FEND becomes

```
FEND    AXT    **, 2
```

As can be seen, the value E3-E1+E2+1 is put into XR2, and XR2 is reduced by E2 every time before the loop statement is executed. The loop is done when <XR2> ≤ E2.

9.17. Linear Address Calculation in FOR Loops.

References to array elements within FOR loops occur very frequently. Without any special method to calculate the addresses of these array elements, it can be very time-consuming to find these addresses each time the loop statement is executed, especially if the subscripts are expressions or the array is of dimension greater than 1.

If the expression for each dimension in a subscripted variable is linear in I (the loop variable of the loop surrounding this reference), then the reference has the property that each time the loop variable I is increased by the fixed amount E2, the address of the subscripted variable is increased by a fixed amount $\emptyset$ (E2).

Linear address calculation, then, refers to the process of calculating the address of each array element referred to in the FOR loop, before the FOR loop is entered (with I.=E1, the initial value); and then increasing each such calculated address by a fixed amount each time the statement of the loop is entered.

Tests have indicated that this method can cut the computing time in half, or even more, depending on the program.

This section indicates exactly under which conditions the linear address calculation can be executed, and how it is done. References are made to the object program of FOR loops, shown in sections 9.16.4. and 9.16.5.

9.17.1. Admissible and proper admissible subscripts.

A subscript is called admissible with I (ADMI) if it is linear in the loop variable, contains only admissible variables and/or constants, and only the operations +, -, *.

A subscript is called admissible (ADM) if it is ADMI but does not contain the loop variable.

A subscript is called proper admissible with I (PADMI) if it is ADMI, but all admissible variables in it are proper admissible for this loop.

A subscript is called proper admissible (PADM) if it is PADMI but does not contain the loop variable.

A subscript is called perfect admissible with I (PEADMI) if it is ADMI and, if it has n dimensions, the expressions for the first n-1 dimensions do not contain the loop variable and the expression for the n^{th} dimension is of the form I, or $I \overset{+}{-}$ expression, where the expression does not contain I. In the same way define perfect proper admissible subscripts (PEPADMI). PEADMI and PEPADMI subscripts are referenced in a special way in perfect FOR loops (see section 9.17.3.). In other FOR loops they are treated like ADMI and PADMI subscripts respectively.

9.17.2. Linearly calculable array elements.

The following table indicates which elements can be calculated linearly. L means calculate the address initially and increment each time through the FOR statement. O indicates calculate the address only initially and use it throughout the execution of the loop.

loop \ array element	ADMI	ADM	PADMI	PADM	PEADMI	PEPADMI	others
perfect	L	O	L	O	L*	L*	no
feasible	L	O	L	O	L	L	no
almost feasible	no	no	L	O	L	L	no
unfeasible	no	no	no	no	no	no	no

*incrementing done differently than others.

9.17.3. Object program for linearly calculable array elements.

If the same array element appears more than once in a FOR loop, the address will be calculated only once and stored in the first location of the object program which references this array element. All other references use this location. An example will illustrate this:

```
A[I+2,J] .= -A[I+2,J];

  B .= A[I+2,J] * C + A[I+2,J];
```

The object program (assuming A[I+2,J] to be linearly calculable) is:

```
A α     CLS     **
        STO*    A α
        LDQ*    A α
        FMP     C
        FAD*    A α
        STO     B
```

The address part of Aα must of course contain the address of the array element A[I+2,J]. This is put in before the loop statement is entered (sections 9.17.4. and 9.17.5.).

All PEADMI and PEPADMI array elements in a perfect FOR loop are referenced a little differently. These are tagged with XR2, and only XR2 is changed before the loop statement is again entered. See section 9.16.6.

If A[I+2,J] is PEADMI or PEPADMI in a perfect FOR loop (J must be the loop variable), then the above object program would appear as:

```
A α     CLS     **,2
        STO*    A α
        LDQ*    A α
        FMP     C
        FAD*    A α
        STO     B
```

The address part of A $\propto$ would contain, for the first time through the loop statement, the address > A[I+2,J] < + E3-E1+1 , since XR2 contains E3-E1+1 initially (section 9.16.6.). Then, each time through the loop, J is increased by E2 and XR2 is decreased by E2.

$\phi_{\text{PEADMI}}(E2) = \phi_{\text{PEPEADMI}}(E2) = E2.$

9.17.4. Subroutines to calculate array element addresses.

Refer to the object program of FOR loops, sections 9.16.5. and 9.16.6.

In order to calculate the increment $\emptyset$ (E2) for an array element address, if the loop variable is incremented by E2, it is sometimes necessary to calculate the array element address for both I.= E1 and I.= E1+E2. Therefore the calculation of an array element address is done by a subroutine.

```
Aβ      SXA     Aβ1, 4

        calculate address, put
        into AC address (see
        section 9.12.)

Aβ1     AXT     **, 4
        TRA     1,4
```

All different array elements which are linearly calculable have such a subroutine in the object program (in the place indicated in section 9.16.5. and 9.16.6.), with the following exceptions:

If a subroutine for A1[E1,...,En] appears in the list, and A2[E1,...,En] occurs in the loop, then no subroutine is necessary for the latter if n = 1, or if A1 and A2 are declared in the same statement: ie

ARRAY ..., A1,...,A2,...[...];

The address for A2[E1,...,En] will be calculated using the address of A1[E1,...,En], as described in section 17.5. It can be seen that

>A2[E1,...,En] < = > A1[E1,...,En] < - > A1[o,...,o] < + > A2[o,...,o] <.

9.17.5. Initialize array element addresses.

This consists of calculating initial addresses for array elements (see 9.17.4.) and calculating ϕ (E2) (the increment when I is increased by E2) for each array element, if necessary. This is broken into two parts. We assume that Aα is the location where the address is to be inserted (see 9.17.3.), and A[E1,...,En] is the array element.

(1) Calculate initial address for element.

(a) PEADMI and PEPADMI array element in a perfect FOR loop.

```
TSX    Aβ, 4
ADD    AUX_S
STA    Aα
```

(subroutine to calculate address)
(see 9.16.6. AUX_S contains the value E3-E1+1)

(b) All other elements

```
TSX    Aβ, 4
STA    Aα
```

Suppose another element, A1[E1,...,En] is referenced. Then if A1 appears in the same declaration as A, or n = 1 (ie. A1[E1] and A[E1]), no subroutine appears for calculating the address of A1[E1,...,En]. Instead the object program looks like

```
TSX    Aβ, 4
STA    Aα
SUB    A+n-1        (location containing > A[o,...,o]<)
ADD    A1+n-1       (location containing > A1[o,..,o]<)
STO    A1α
```

See sections 9.5.1. and 9.17.4.

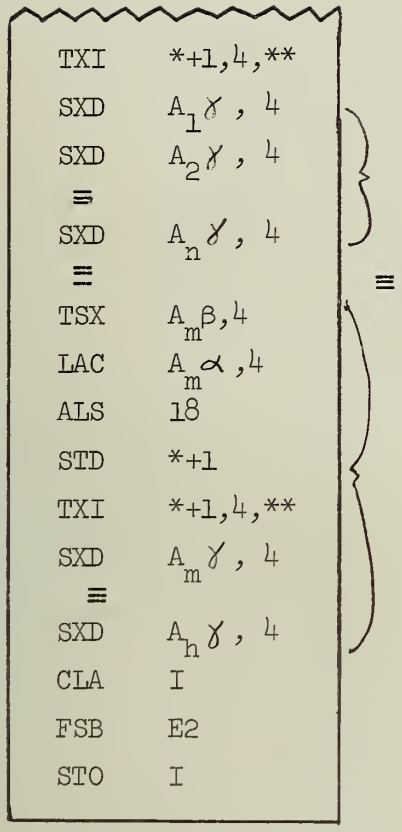
(2) PEADMI and PEPADMI array elements in perfect FOR loops are incremented with XR2. (ϕ(E2) = E2 for such elements). PADM and ADM elements need no incrementing, since they do not depend on the loop variable. The following, then, refers to all other cases.

Directly after the object program defined in 9.17.5.(1) comes the object program for calculating ϕ(E2) for each array element. As shown in 9.17.6., incrementing is done by

```
        LXA   Aα, 4
A )'    TXI   *+1, 4, φ (E2)
        SXA   Aα, 4
```

The object program for calculating each ϕ (E2) is

```
CLA     I          Get set for I:= E1+E2 to
FAD     E2         calculate each subscript again
STO     I
TSX     A1β,4      Go calculate address
LAC     A1,α,4     substract address calculated with
ALS     18         I := E1 from address calculated
STD     *+1        with I:= E1+E2
```



store difference ϕ (E2)

store this difference to help increment all addresses known to have the same ϕ (E2)

calculate and store ϕ (E2) for another set of array elements

reset I to E1

If no such calculation appears, the increasing and decreasing of I is omitted.

As many SXD's appear as different array elements which are known to have the same ϕ (E2), as described in 9.17.5.(1) (b).

9.17.6. Incrementing array element addresses.

For each array element which must be incremented, the instructions

```
        LXA     Aα,4
Aγ      TXI     *+1,4,ϕ (E2)
        SXA     Aα,4
```

appear. Where they go can be seen in the object program for FOR loops, sections 9.16.5. and 9.16.6.

9.18. Object Program Examples.

9.18.1. Perfect loop.

Source program:

```
FOR I:=1 STEP 1 UNTIL N DO
  BEGIN A[I]:= A[2*I] - A[I] + A[I-1];
        B[I]:= A[I-1] +B[2*I];
  END;
```

The object program consists of 80 instructions, 63 of which are initialization. The loop statement and test comprise only 17 instructions.

```
            CLA     =1.o
            STO     I
            CLA     N                 (with char. = (233)8)
            UFA     CONSTLOC          <CONSTLOC> = -E1+1=o
            STO     AUXS              SAVE E3-E1+1 fixed point
            UFA     =1.o              ADD E2
            TMI     F2                E3 < E1 -DON'T ENTER LOOP
            PAX     o,2               E3+E2-E1+1 → XR2
            TRA     FINIT             GO INITIALIZE
STATE.      BSS     o
A1α         CLA     **                A[2*I]
A2α         FSB     **,2              A[I]
A3α         FAD     **,2              A[I-1]
            STO*    A2α               A[I]
            CLA*    A3α               A[I-1]
B1α         FAD     **                B[2*I]
B2α         STO     **,2              B[I]
            CLA     I                 INCREASE
            FAD     =1.o              LOOP
            STO     I                 VARIABLE
            LXA     A1α, 4            INCREMENT ADDRESSES
A1α         TXI     *+1,4,**
            SXA     A1α, 4
            LXA     B1α, 4
```

Date:	9/28/64
Section:	9.18.1.
Page:	1 of 3
Change:	1

```
B1      TXI     *+1,4,**
        SXA     B1α, 4
        TIX     STATE.,2,1      DECREASE, TEST END OF LOOP
F2      TRA     FEND            DONE WITH LOOP
A1β     SXA     A1β1,4
        LDQ     =2.o
        FMP     I               subroutine for calculating
        UFA     .ZER            > A[2*I] <
        ALS     lo
        ARS     lo
        ADD     A
A1β1    AXT     **,4
        TRA     1,4
A2β     SXA     A2β1,4
        CLA     I
        UFA     .ZER            subroutine to
        ALS     lo              calculate > A[I]<
        ARS     lo
        ADD     A
A2β1    AXT     **,4
        TRA     1,4
A3β     SXA     A3β1,4
        CLA     I               subroutine to
        FSB     =1.o            calculate  > A[I-1]<
        UFA     .ZER
        ALS     lo
        ARS     lo
        ADD     A
A3β1    AXT     **,4
        TRA     1,4
FINIT   TSX     A1β,4           initialize addresses
        STA     A1α               >A[2*I]<
        SUB     A
```

Date:	9/28/64
Section:	9.18.1.
Page:	2 of 3
Change:	1

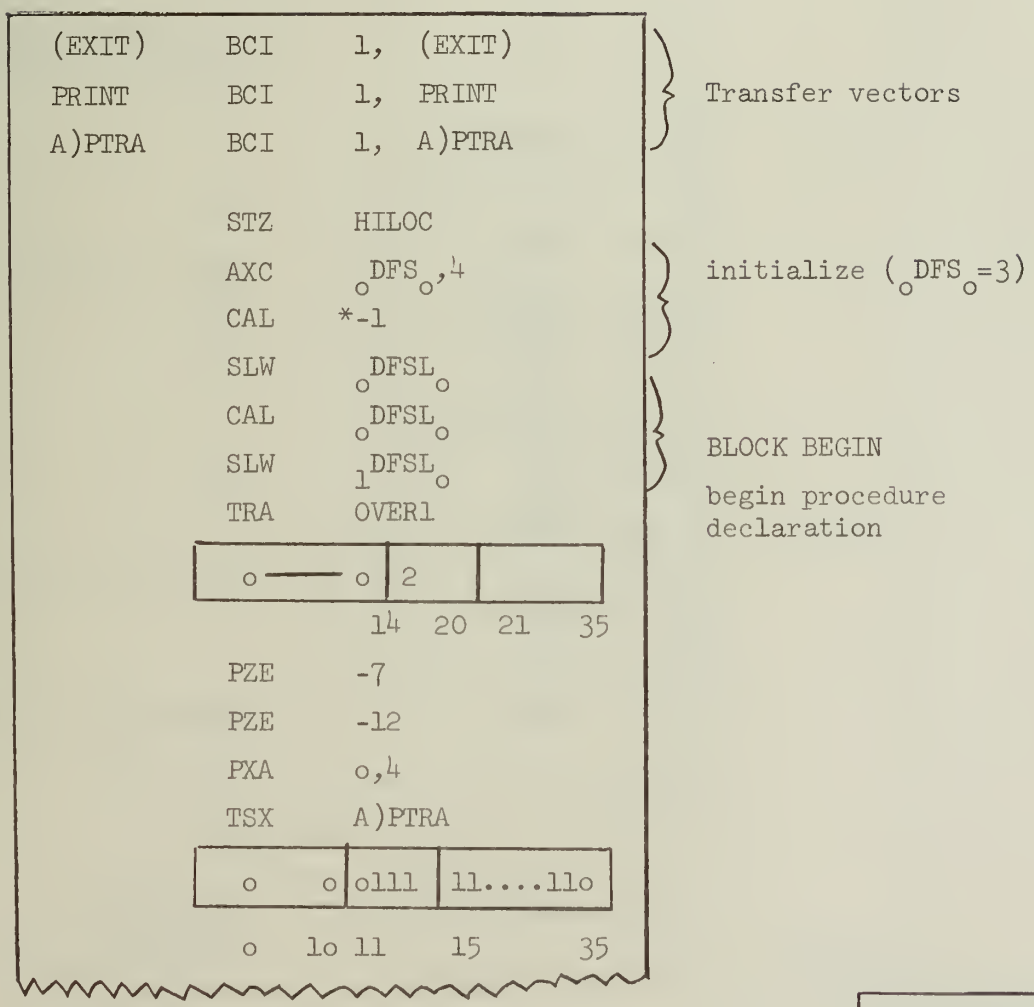
```
        ADD     B
        STA     B1 α            > B[2*I]<
        TSX     A2β,4
        ADD     AUXS            =E3-E1+1
        STA     A2α             > A[I] < +E3-E1+1
        SUB     A
        ADD     B
        STA     B2α             > B[I] < +E3-E1+1
        TSX     A3β,4
        ADD     AUXS
        STA     A3α             > A[I-1] < +E3-E1+1
        CLA     I
        FAD     =1.o            increase I
        STO     I
        TSX     A1β,4           get ∅ (E2) for
        LAC     A1α,4           A[2*I]
        ALS     18
        STD     *+1
        TXI     *+1,4,**
        SXD     A1γ,4           ∅ (E2) stored for A[2*I]
        SXD     B1γ,4           ∅ (E2) stored for B[2*I]
        CLA     I               reset I
        FSB     =1.o
        STO     I
        TIX     STATE.,2,1      TEST, JUMP TO STATEMENT
FEND    BSS     o               DONE
```

9.18.2. Procedure and procedure call.

Source program

```
BEGIN  PROCEDURE  A(I,C);
       VALUE I;  REAL C;  INTEGER I;
       BEGIN C.= I*C;
       END  A;
       REAL  D;
       D.= 1;
       A (3*2o,D);
       PRINT (D);
END;
```

Object program



Date:	9/28/64
Section:	9.18.2.
Page:	1 of 3
Change:	1

```
            LDI     oDFSL1, 1       Begin procedure body
            XEC     P2, 1           Get address of C
            NOP     -1,1
            LDQ     I,1
            FMP     o,4             I*C
            STO     AUX,1
            XEC     P2, 1           Get address of C
            NOP     -1,1
            CLA     AUX,1
            STO     o,4             STORE IN C
            XEC     -2,1            RESTORE XR2
            LDI     -5,1            RESTORE DFS
            LDQ     -3,1            THE RETURN TRANSFER
            STQ     *+2
            XEC     -1,1            RESTORE XR1  (FFS)
            PZE                     RETURN
OVER1       CLA     CONST1+5
            STO     D               D:=1
            TRA     OVER2
TH1         LDQ     CONST1+6        THUNK FOR 3*2o
            FMP     CONST1+7
            STO     .EXTRA
            AXC     .EXTRA,4
            TRA     2,2
OVER2       LDI     1DFSLo
            TSX     A,4             CALL procedure A
t1          TSX     TH1,2
t2          AXC     D,4
            TSX     PRINT,4         RETURN POINT, PRINT D
            STR
            STR     D
            STR     o
            TSX     (EXIT) ,4       END PROGRAM
CONST1      PZE                     CONSTANT LIST
```

```
                OCT     77...7
                OCT     233o...o
                OCT     1
.EXTRA          PZE
                DEC     1.o
                DEC     3.o
                DEC     2o.o
```

ffs_o

D	
$_1DFSL_o$	$_1DFSL_o$
$_oDFSL_o$	$_oDFS_o$ = HILOC - 3

FFS_o = HILOC

ffs_1 (when in procedure A)

AUX	5 locations
P_2	AXC C,4
P_1	6o.o
$_1DFSL_1$	AXC $_1DFS_1$, 4
	AXC HILOC-3,4
$_oDFSL_1$	AXC $_oDFS_1$,4
	TRA t_2+1
	AXT ,2
	AXT **,1

FFS_1

Date:	9/28/64
Section:	9.18.2.
Page:	3 of 3
Change:	1

9.19. Index of Definitions and Conventions.

A P P E N D I C E S

Appendix A: Bibiliography 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), 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.

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

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 Representation	Tolerated Hardware Representation
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	
true	Boolean true	'TRUE'	
false	Boolean false	'FALSE'	
+	plus sign	+	
-	minus sign	-	
×	multiplication sign	*	
/	division sign	/	
÷	integer division sign	//	
↑	exponentiation	'PØWER'	**
<	less than	'LESS'	'LS'
⩽	less than or equal to	'NØT GREATER'	'LQ'
=	equal to	'EQUAL'	'EQ'
⩾	greater than or equal to	'NØT LESS'	'GQ'
>	greater than	'GREATER'	'GR'
≠	not equal to	'NØT EQUAL'	'NQ'
≡	logical equivalent	'EQUIV'	'EQV'
⊃	logical implics	'IMPL'	'IMP'
∨	logical or	'ØR'	
∧	logical and	'AND'	
¬	logical negation	'NØT'	
go to		'GØ TØ'	
if		'IF'	
then		'THEN'	
else		'ELSE'	
for		'FØR'	
do		'DØ'	
,	comma	,	
.	decimal point	.	
10	base 10	' (apostrophe)	
:	colon	..	
;	semi-colon	.,	
:=	assignment sign	.=	
# or b	blank space		
step		'STEP'	
until		'UNTIL'	
while		'WHILE'	
comment		'CØMMENT'	
(	left parenthesis	(	
)	right parenthesis	)	
[	left bracket	(/	
]	right bracket	/)	
‘	left string quote	'('	
’	right string quote	')'	
begin		'BEGIN'	
end		'END'	
own		'ØWN'	
boolean		'BØØLEAN'	
integer		'INTEGER'	
real		'REAL'	
array		'ARRAY'	
switch		'SWITCH'	
procedure		'PRØCEDURE'	
string		'STRING'	
label		'LABEL'	
value		'VALUE'	
code		'CODE'	
finis		'FINIS'	

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

Appendix D.

Examples.

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
```

```
'PRØCEDURE' SPUR (A) ØRDER..(N) RESULT..(S).,
     'VALUE' N., 'ARRAY' A., 'INTEGER' N., 'REAL' S.,
     'BEGIN' 'INTEGER' K.,
          S.= 0.,
          'FØR' K .= 1 'STEP' 1 'UNTIL' N 'DØ'
          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

'PRØCEDURE' TRANSPØSE (A) ØRDER .. (N)., 'VALUE' N.,
     'ARRAY' A., 'INTEGER' N.,
          'BEGIN' 'REAL' W., 'INTEGER' I, K.,
               'FØR' I.= 1 'STEP' 1 'UNTIL' N 'DØ'
                    'FOR' K.= 1 + I 'STEP' 1 'UNTIL' N 'DØ'
                         'BEGIN' W.= A (/ I, K/).,
                              A (/I, K/). = A(/K, I/).,
                              A (/K, I/). = W
                         'END'.'
          'END' TRANSPØSE
'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
```

Date:	6/1/64
Section:	A-D
Page:	3 of 6
Change:	1

Example 3., continued.

```
'PRØCEDURE' EULER (FCT, SUM, EPS, TIM)., 'VALUE' EPS, TIM.,
'INTEGER' TIM., 'REAL' 'PROCEDURE' FCT., 'REAL' SUM, EPS.,
'CØMMENT' EULER CØMPUTES THE SUM ØF FCT (I) FØR
I FROM ZERØ UP TØ INFINITY BY MEANS ØF A
SUITABLY REFINED EULER TRANSFØRMATIØN.,
'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).,
          'FØR' K.= 0 'STEP' 1 'UNTIL' N 'DØ'
               'BEGIN' MP.= (MN + M (/K/))/2., M(/K/).=MN.,
                  MN.= MP
               'END' MEANS.,
          'IF' (ABS (MN) 'LS' ABS (M (/N/) )) 'AND' (N 'LS' 15) 'THEN'
               'BEGIN' DS.= MN/2., N.= N + 1.,
                  M (/N/).=MN
               'END' ACCEPT                                    'ELSE'
               DS.= MN.,
               SUM.= SUM + DS.,
          'IF' ABS (DS) 'LS' EPS 'THEN' T .= T + 1 'ELSE' T .= 0.,
          'IF' T 'LS' TIM 'THEN' 'GO TO' NEXT TERM
'END' EULER
'FINIS'
```

Example 4.

```
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) ≥ 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 complexdiv;
     read (r, p, q, s); complexdiv (r, p, q, s, t, u); print (r, p, q, s,
     t, u)
end
```

Date:	6/1/64
Section:	A-D
Page:	5 of 6
Change:	1

Example 4., continued.

```
$ ALGØL
$ GØ
'BEGIN' 'CØMMENT' CØMPLEX DIVISIØN USING ALGØRITHM 116,
CØMMUNICATIØNS ØF ACM, AUG. 1962.,
     'REAL' R, P, Q, S, T, U., 'INTEGER' N.,
     'PRØCEDURE' CØMPLEXDIV (A, B, C, D) RESULTS.. (E, F).,
     'VALUE' A, B, C, D., 'REAL' A, B, C, D.,
     'CØMMENT' CØMPLEXDIV YIELDS THE CØMPLEX QUØTIENT ØF
     A + IB DIVIDED BY C + ID.,
     'BEGIN' 'REAL' R, DEN.,
          'IF' ABS (C) 'NØT 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' CØMPLEX DIV.,
     READ (R, P, Q, S).,
     CØMPLEX DIV (R, P, Q, S, T, U).,
     PRINT (R, P, Q, S, T, U)
'END'
'FINIS'
```