Report No. 1099

AN EXTENDED ALGOL 60 COMPILER

for the

UNIVAC 1108

STAFF

Andrew R. Jennings Computing Center

Case Western Reserve University

April 1968

P R E F A C E

This manual is intended to be a definitive user's guide to an Algol 60 compiler that has had five years' extensive development and use in the Andrew R. Jennings Computing Center at Case Western Reserve University. In the initial design stages it was decided to have the compiler adhere as close to the "Revised Report on the Algorithmic Language Algol 60" as possible without at the same time excluding the possibilities of extending the language for the user's benefit. Thus, the reader will discover that substantial additions have been made to the language (e.g. a sort/merge facility) while very few deviations have been made from the formal report. He will also note that quite comprehensive and understandable run-time diagnostics have been included in this implementation.

Naturally both the program and this document itself are the end products of the efforts of many people -- in particular we single out Mr. Paul Meiland who "volunteered" to produce this manual. Following is an alphabetical list of those who have had major responsibilities in creating the program and/or the manual:

David Abt
H. Lynn Beus
Martin Charns
John Dedourek
Stan Eisenstadt
Gilbert Hansen
George Haynam
Nicholas Hubacker
Robert Ladner
W. C. Lynch
John Massie
Paul Meiland
Julius Nadas
Frank J. Olynyk
Sharon Sanford
David Santich
Jon Shomer
Joseph Speroni

Truth Walkey

Kenneth Walter

Frederick Way

James Wilt

Five Thousand Users

In summary, any criticisms, complaints and/or objections to this work should be directed to Prof. F. Way, III, Associate Director, Computing Center -- while any complimentary remarks, praises and/or general enthusiasm should be directed to any of the individuals on the above list.

This work was supported in part by National Science Foundation (NSF GP-642).

TABLE OF CONTENTS

GENERAL INTRODUCTION

The first part of this manual is intended to give an introduction to the uninitiated into the process of using a computer to solve problems. No assumptions will be made about "mathematical maturity" and no knowledge about computers will be assumed of the reader. The really difficult part of explaining a problem solution process to a computer is that one must be able to make the explanation in such a way that a person (or the computer) who is totally unfamiliar with the problem will be able to follow your recipe exactly and obtain the desired results. One device for exhibiting recipes to either computers or other people is to draw flow charts. Again, the reader is cautioned that the really difficult part of using the computer will usually turn out to be the construction of a valid flow chart for the problem.

EXAMPLE: Suppose we want to write down a set of instructions for someone to follow in order to solve the following problem:

> Given numerical values of A, B, C find the numerical value of X which satisfies the equation

$$Ax^2 + Bx + C = 0 .$$

At this point the reader yawns and says the solution is obvious, namely

$$x = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A} .$$

As it turns out, this is NOT a solution to the problem we are considering! Why not? Because we were supposed to write down a set of instructions for solving the problem and the recipe above will not be of any use at all unless the person reading the recipe has already done some problems like this, has used this recipe, and understood what was behind it. This set of requirements is asking a little too much from any computer. They will all have already solved this type of problem, but it is very unlikely that any of them understood what they were doing!

One simple method of making sure that we get the results desired is to admit at the outset that the computer is not particularly bright, but is quite rapid. Therefore we will supply it with all sorts of advice and instructions and let it proceed on its stupid but rapid way to the problem solution. For the problem at hand we know that a solution consists of two roots. However the specification of a root involves both a real part and an imaginary part. Therefore we must recognize that a complete set of answers requires writing down four real numbers.

At this point the reader is urged to examine Figure 1. Begin at the place labeled START and follow the arrows. You should be able to convince yourself that no matter what values of A, B, C are used that this flow chart will take care of the situation.

If we can construct such a flow chart for any particular problem, then the process of using the computer to solve the problem is fairly trivial. All we need do at this point is to tell the computer what the flow chart looks like and set it to work. The purpose of the first part of this manual is to describe a rather elegant (and not too complicated) method for telling the computer what the flow chart looks like. In other words we translate the flow chart into some other language which the computer will understand (not really understand, but at least accept). The language at hand is ALGOL-60.

The next batch of information will be much easier to understand if the reader will arm himself with a pencil and scratch paper and follow directions!

All of the instructions and advice which we are about to give the computer must be written using only the capital letters of the English alphabet, the ten decimal digits and some assorted punctuation marks. In addition there are some words and abbreviations which have special meaning to the computer and may NOT be used for any use other than what the computer thinks they are supposed to mean. Any time such a special word is used in this manual it will be underlined to call to your attention that this is a RESERVED WORD. The input to the computer will NOT be underlined, but the machine already knows all of the reserved words and will not make mistakes with them.

Keeping an eye on the flow chart we now start writing a program: The reader is advised to write the program on his scratch pad as we go. First program line:

COMMENT THIS IS A QUADRATIC EQUATION SOLVER EXAMPLE $

FLOW CHART for solution of $AX^2 + BX + C = 0$

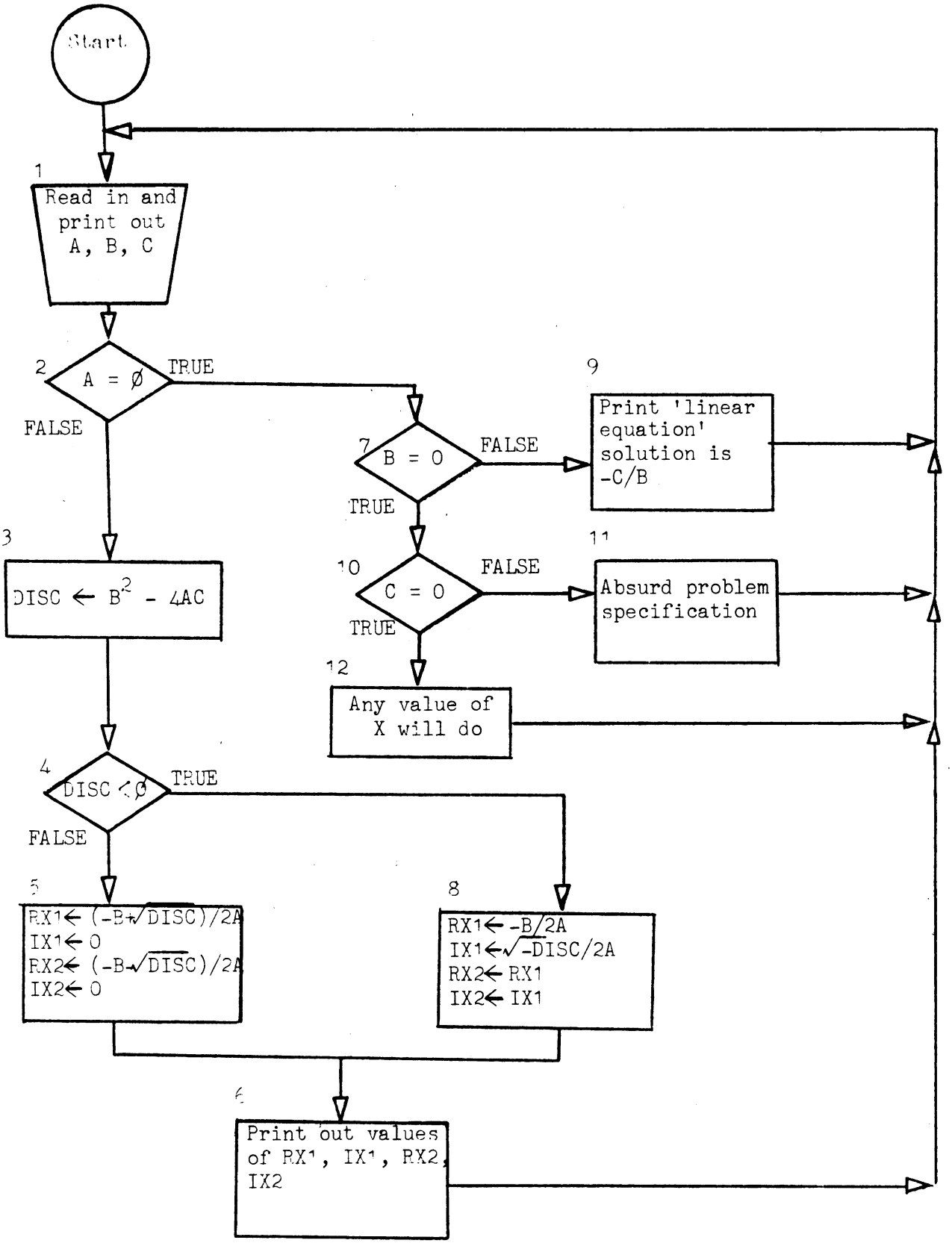


FIG. 1

Note that the word COMMENT is underlined, i.e. it is a reserved word. The computer will ignore any characters to the right of the word COMMENT until it encounters a dollar sign. The reason for including COMMENT in the language is so that the human will have a chance to see what is going on in the program, not only while he is writing it, but six months later when he once again comes across it.
Box 1 in the flow chart says to obtain and record values for A, B, C.

Next program line:

READ(A,B,C) $

Again note the underlined word READ; it has a special meaning. The A,B,C mean that the computer is to read in a card and accept the first three numerical values on the card as being the values for A,B,C respectively.

Next program line:

WRITE('QUADRATIC EQUATION WITH A, B, C RESPECTIVELY, ',A,B,C) $

The lines of program are punched on cards exactly as shown -- including everything, spaces, punctuation marks and all -- especially the dollar sign! The line which we have just written will cause the computer to print out the message included between the quotation marks, followed by the three numerical values which it read in from the data card.

We are now at Box 2 in the flow chart. Notice that it effectively says that if A is not equal to zero we should go to Box 3.

Next program line:

IF A NEQ 0 THEN GO TO B3 $

The only non-obvious part of this line is just what B3 is supposed to mean. We will take care of this when we get to the place where Box 3 is actually used in the program. Now the reader may well be a little curious about what happens if the proposition " A NEQ 0 " is NOT true. In this case the computer SKIPS the instruction which says " GO TO B3 ", and goes on in the program to see what to do next. In this flow chart Box 7 comes next in the case where " A NEQ 0 " is false (i.e. A is equal to zero).

Next program line:

```
IF B NEQ 0 THEN GO TO B9 $
```

Applying the same reasoning as used in the last paragraph we see that there is nothing particularly complicated about what the computer action will be in either case (i.e. the cases where 1. " B NEQ 0 " is true and 2. is false).

We are now at Box 10.
Next program line:

```
IF C NEQ 0 THEN GO TO B11   $
                    WRITE('NO PROBLEM SPECIFIED')  $ GO TO B1  $
```

The reader now looks at the program he has written and spots a problem --- he knows where B1 is, but has not told the computer about where B1 is supposed to be. We now go back and CHANGE THE SECOND LINE OF THE PROGRAM so that it reads:

```
B1..READ(A,B,C)  $
```

This change in effect defines what B1 means. Note the use of the two periods following the label "B1"; the periods tell the machine that B1 is a label which is being defined (i.e. placed) at this point. Note that the possibility of placing a label at this point (before the READ) was (is now) fairly obvious since there are two lines on the flow chart which enter the first box. We now start to clean up the remaining odds and ends of the flow chart.

Next program line(s):

```
B11.. WRITE('NOT A VERY REASONABLE SPECIFICATION') $  GO TO B1  $
B9..  WRITE('LINEAR EQUATION, SOLUTION IS X=',-C/B) $  GO TO B1  $
```

Note that the last line above prints out the message between the quotation marks followed by the numerical value of -C/B.

Continuing at box 3:

```
B3..  DISC = B**2 - 4*A*C  $
```

Note that the above line says "the present value of DISC is replaced by the result of computing B squared minus 4 times A times C". The equal sign always means "is replaced by".

Continuing at box 4:

```
IF DISC LSS 0 THEN BEGIN
     RX1 = -B/(2*A)  $   IX1 = SQRT(-DISC)/(2*A)  $
     RX2 = RX1  $
     IX2 = -IX1   END
           ELSE
     BEGIN
     RX1 = (-B + SQRT(DISC))/(2*A)  $   IX1 = 0  $
     RX2 = (-B-SQRT(DISC))/(2*A) $  IX2 = 0  END  $
```

We have now encountered several new items -- for instance SQRT, the meaning of which is not at all mysterious -- however, note that the quantity which is to have its square root extracted must be enclosed in parentheses. Other new items turn out to be the use of BEGIN and END which are used to enclose several statements into one lump to make it clear to the computer just how much to do or skip as the case may be. We have so far encountered two forms of construction which both started with the word IF, they are:

IF <Boolean expression> THEN <unconditional statement>

and

IF <Boolean expression> THEN <unconditional statement> ELSE <statemer

The use above of the marks "<" and ">" means that the text between the < and > refers to a general construction which can be (and is) defined for ALGOL-60. A Boolean expression is something which has a value of either true or false (e.g. A NEQ 0). An unconditional statement is one that does NOT start with the word IF.

This is a rather negative definition of the idea of an unconditional statement, but it will do for the time being. We may now be tempted to suppose that a conditional statement is one which does start with the word IF, which is correct. The construction used in the program which started with the word BEGIN, and ended with the word END, is an example of an unconditional statement (note that it did NOT start with the word IF).

The action of the first construction for IF above is to test the truth of the Boolean expression and then to either do or skip the following unconditional statement (which follows the word THEN).

The action of the second construction for IF is again to test the truth of the Boolean expression and then to do either the unconditional statement following the word THEN or to do the statement following the word ELSE, but never to do both and never to skip both.

Next program line:

```
WRITE('REAL X1, IMAG X1, REAL X2, IMAG X2 ',RX1,IX1,RX2,IX2)  $
GO TO B1  $
```

At this point our program is almost done, but not quite!

With a problem of the type presently under discussion it is a rather simple matter to test all of the alternative possibilities for the data. We now furnish some data to test our program. The data cards are the cards which will be read when the computer is executing the program and encounters the word READ. Since these cards are to be read during the execution phase of the program they MUST (obviously) follow the last card of the program itself.

Next set of cards (one for each line)

```
2    5    2
1   -2   -15
1   -4    13
0    3    18
0    0     2
0    0     0
```

There is now only one more thing to do in order to have a complete program. As we will later discover, it is always necessary to specify "types" of variables. The only reasonable method at the moment is to specify all of the variables as of type REAL. We do this by inserting one line at the very top of the program which reads:

```
REAL  A,B,C,DISC,RX1,IX1,RX2,IX2  $
```

If you have followed directions you are to be congratulated! You have now written a complete ALGOL-60 program which should operate correctly. Your scratch paper should now have the following information on it:

```
      REAL  A,B,C,DISC,RX1,IX1,RX2,IX2  $
      COMMENT THIS IS A QUADRATIC EQUATION SOLVER EXAMPLE  $
B1..  READ(A,B,C) $
      WRITE('QUADRATIC EQUATION WITH A, B, C RESPECTIVELY, ',A,B,C) $
      IF A NEQ 0 THEN GO TO B3  $
      IF B NEQ 0 THEN GO TO B9  $
      IF C NEQ 0 THEN GO TO B11  $
      WRITE('NO PROBLEM SPECIFIED') $ GO TO B1  $
      B11..WRITE('NOT A VERY REASONABLE SPECIFICATION')  $  GO TO B1  $
      B9..WRITE('LINEAR EQUATION, SOLUTION IS X=',-C/B)  $  GO TO B1  $
      B3..  DISC = B**2 - 4*A*C  $
      IF DISC LSS 0 THEN BEGIN
           RX1 = -B/(2*A)  $     IX1 = SQRT(-DISC)/(2*A)  $
           RX2 = RX1  $
           IX2 = -IX1   END
                ELSE
           BEGIN
           RX1 = (-B + SQRT(DISC))/(2*A)  $   IX1 = 0  $
           RX2 = (-B-SQRT(DISC))/(2*A)  $  IX2 = 0  END  $
      WRITE('REAL X1, IMAG X1, REAL X2, IMAG X2, ',RX1,IX1,RX2,IX2)  $
      GO TO B1  $
```

The data cards could appear as follows:

```
2     5     2
1    -2   -15
1    -4    13
0     3    18
0     0     2
0     0     0
```

We once again call your attention to the fact that the underlines are used only to call the reader's attention to the fact that some of the words are reserved for special meanings. The input which goes into the computer will NOT be underlined -- it "knows" which are reserved words and which are not. For example:

The fifth line of the program will appear to the computer as:

IF A NEQ 0 THEN GO TO B3 $

Note that if we smash the characters together:

IFANEQ0THENGOTOB3$

then we cannot make any sense out of the line and as it turns out -- neither can the computer. Thus it would appear that spaces are not only useful - but absolutely necessary. The easiest view to take of the rules regarding spaces is that if you would accept a particular rendering of a program from your secretary, then the computer will very likely also accept that form. If you cannot easily decide where things (words, etc.) start and end, then the computer will not only not be able to decide but it won't even make an attempt. The rules for the use of spaces are not as inflexible as they might appear -- if one space is necessary, then any number will be acceptable (i.e. at least one), if no spaces are NECESSARY then spaces will not hurt anything.

For example:

RX2=RX1$

is completely equivalent to

RX2 = RX1 $

BUT the following line will not do at all:

R X 2 = R X 1 $

It would seem then, that there are some more rules with regard to how one names variables. The rules for variable (and label) names are quite simple.

1. Names must start with an alphabetic character and may contain only alphabetics and numerics -- never spaces or punctuation marks.
2. The computer examines only the first twelve characters of a name to "remember" the name, thus

HEMIDEMISEMIQUAVER

is a valid name but is regarded as equivalent to

HEMIDEMISEMIANTIDISESTABLISHMENTARIANISM

since the first twelve characters are the same in both.

A consequence of the rules above is that although

RX2

is a legitimate name for a variable,

is NOT an acceptable name for a variable. (It does not start with an alphabetic character).

There is now some explanation owed to the reader as to how the computer decides which arithmetic operation to perform first if there is any choice. For example

-B/2*A	might mean	-(B/2*A)
	or	-B/(2*A)
	or	(-B/2)*A

or possibly some other things, the question being not what it might mean but what it does mean.

Common usage suggests that we would like to have our arithmetic expressions evaluated by doing in order

1. Exponentiation (**) and
   Unary minus (the kind of minus that does not have a number or a variable to its left) on an equal basis, but evaluate the unary minus from right to left.
2. Multiplication and/or division
3. Addition and/or subtraction

with all of the above (except the unary minus) being done on a left to right basis at any particular parenthesis level. These rules result in:

Examples:

Desired Quantity	Algol-60 Representation
Q^{-2}	Q**-2
B^2-4AC	B**2-4*A*C
$-G^{-2}-H^{-5}$	-G**-2-H**-5
$\frac{1}{2}GT^2$	G*T**2/2
$A(B^{-(2^{-B})})(C^{-3})$	A*B**-2**-B*C**-3

The last example above is correct, but points up a rule that should not be ignored, namely--

"WHEN IN DOUBT, USE PARENTHESES BY THE BUSHEL"

Thus, if you desire

$$A(B^{-(2^{-B})})(C^{-3})$$

write

```
A*(B**-(2**-B))*(C**-3)
```

CARD PUNCHING RULES

The rules for punching the program on cards are quite simple:

1. The computer examines columns 1-72 only -- all else is ignored.
2. The computer views column 72 on one card as being directly to the left of column 1 on the following card.
3. The user may put any valid information at all into cols. 73-80, e.g., the name of the program.

These rules imply that the machine does not know and in fact does not care whether statements exist one or more than one per card. The user is advised to generally restrict himself to one statement per card in order to make his life easier when the time comes for making alterations in the program.

The data cards have a separate set of rules:

1. The computer examines columns 1-80 for data
2. In no case may a number be split between two cards, since the machine DOES NOT consider column 80 on one card to be connected to column 1 on the following card.

A description of the Exec III control cards that are used to actually run a program through the computer is given in Chapter XIV, USING ALGOL UNDER EXEC III.

I...

INTRODUCTION

This section is intended as a reference manual in the use of an extended Algol 60 language, based in part on the "Revised Report on the Algorithmic Language ALGOL 60" (Communications of the ACM, Vol. 6, January 1963, 1-17.) The Algol translator is a program which accepts statements expressed in the Algol language and produces machine-language programs for the Univac 1107 Thin Film Memory Computer.

The Algol translator (compiler) utilizes a Univac 1107 with 65,536 words of core memory, two FH-880 magnetic drums, card readers and line printers. The Algol compiler is an integral part of the Exec III operating system.

The text of this reference manual consists principally of definitions and rules for the use of the translator, examples of these rules, and some sample programs. A set of appendices summarizes the text and lists some details on the operation of the program, the contents of the library, etc.

Whenever a term is defined, it is underlined in the defining sentence. Greek letters or names enclosed in corner brackets (e.g. <integer>) are used in the text to denote generic representations; for example, ϵ is used to represent an expression and Σ to represent a statement. For the most part, other symbols represent themselves.

The examples, which have been used quite liberally, have been employed for "definitions by example" in only those few cases where a formal description has proved particularly unwieldy.

A program that is to be run on a computer must take the form of machine language, i.e., instructions that can be directly decoded and executed by the electronic apparatus of the computer. On the other hand, a program that is to be practical for solving problems should be in a form that is easily written and understood by human beings.

At present there is no programming language that meets both these requirements. Therefore the use of a language such as Algol in computing

involves two closely related but distinct program forms.

The first, called a source program, consists of Algol instructions written by the programmer to describe the process he wants carried out. The second form, called an object program consists of machine language instructions appropriate to the specific computer at hand. In the present case the intermediary that translates source language to machine language is called an Algol compiler.

The compiler is an elaborate program that accepts an Algol source program as input and produces a corresponding machine language program (object program) as output. The object program may then be directly carried out by the computer or it may be stored for later use.

A typical run of an Algol program is as follows:

1) The Algol source program is fed into the computer on punched cards.
2) The Algol compiler translates the program into machine language, printing out the source instructions, some diagnostic messages and, possibly, some error messages. The resulting object program is temporarily stored on the magnetic drum. At this point the compiler's work is done.
3) The object program and any auxiliary routines that it calls for (e.g., SIN, READ, SORT) are copied from the drum to the memory unit (i.e., allocated) and executed by the computer, using data provided by the user on punched cards. Every Algol program uses at least a few auxiliary routines from the Algol system library, which is always stored on the magnetic drum.

Among the many features made available to the programmer by Case Algol are extensive data processing facilities, double precision and complex arithmetic, and bit manipulation. There are also available routines to store programs and data on tape, create visual displays using a plotter, and a sort-merge package.

Deviations From Algol 60

For those familiar with the Algol 60 publication language, the deviations of Case Algol from Algol 60 can be summarized as follows:

1) Uniqueness of an identifier is determined by examining its first twelve characters only (for exceptions, see IDENTIFIERS).

2) Every formal parameter must be mentioned in the specification part of the procedure heading.

3) Numeric labels are excluded from the language.

4) A comma is the only acceptable parameter delimiter in procedure calls.

5) The result of integer exponentiation (I**J, I and J both integer) is always integer.

6) Forward referenced identifiers should be declared by a local declaration.

7) Arrays declared OWN are not dynamically allocated.

8) In a FOR statement using the STEP-UNTIL <for list element>, the step expression is evaluated once for each time the loop is entered.

These and other restrictions are covered in more detail in other sections of this manual.

Many extensions have been made to the Algol 60 language to facilitate the handling of large complex programs. These extensions include:

1) Input/Output routines to provide very flexible handling of various data forms;

2) Double precision and complex arithmetic to extend the scope of scientific computations;

3) General character string operations to provide very flexible data processing features;

4) Provisions for allowing procedures written in Algol or Sleuth or subroutines written in Fortran to be linked to and executed in conjunction with the object program;

5) Options to facilitate debugging of the program;

6) A set of intrinsic functions that allow partial word and bit operations.

BASIC SYMBOLS

The following correspondences are made for the representation of basic symbols:

Basic Symbol for Publication Language	Basic Symbol for Translator	Card Code
true	TRUE	reserved identifier
false	FALSE	reserved identifier
+	+	12
-	-	11
x	*	11-4-8
/	/	0-1
÷	//	0-1 0-1
↑	**	11-4-8 11-4-8
<	LSS	reserved identifier
≤	LEQ	reserved identifier
=	EQL	reserved identifier
≥	GEQ	reserved identifier
>	GTR	reserved identifier
≠	NEQ	reserved identifier
≡	EQUIV	reserved identifier
⊃	IMPL	reserved identifier
∨	OR	reserved identifier
∧	AND	reserved identifier
¬	NOT	reserved identifier
go to	GOTO or GO TO	reserved identifier
if	IF	reserved identifier
then	THEN	reserved identifier
else	ELSE	reserved identifier
for	FOR	reserved identifier
do	DO	reserved identifier
,	,	0-3-8
.	.	12-3-8
&	&	2-8
:	: or ..	5-8 or 12-3-8 12-3-8
;	$ or ;	11-3-8 or 11-6-8
:=	= or :=	3-8 or 5-8 3-8
␣	△	(blank)

Basic Symbol for Publication Language	Basic Symbol for Translator	Card Code
step	STEP	reserved identifier
until	UNTIL	reserved identifier
while	WHILE	reserved identifier
comment	COMMENT	reserved identifier
(	(	0-4-8
)	)	12-4-8
[	(or [	0-4-8 or 12-5-8
]	) or]	12-4-8 or 11-5-8
‘	'	4-8
’	'	4-8
begin	BEGIN	reserved identifier
end	END	reserved identifier
own	OWN	reserved identifier
Boolean	BOOLEAN	reserved identifier
integer	INTEGER	reserved identifier
real	REAL	reserved identifier
array	ARRAY	reserved identifier
switch	SWITCH	reserved identifier
procedure	PROCEDURE	reserved identifier
string	STRING	reserved identifier
label	LABEL	reserved identifier
value	VALUE	reserved identifier

In order to extend the language the following basic symbols have been introduced:

Basic Symbol for Translator	Card Code
TO	reserved identifier
REAL2	reserved identifier
LIST	reserved identifier
FORMAT	reserved identifier
EXTERNAL	reserved identifier

Basic Symbol for Translator	Card Code
COMPLEX	reserved identifier
LOCAL	reserved identifier
DEFINE	reserved identifier
STRING	reserved identifier
XOR	reserved identifier
GO	reserved identifier
TRACE	reserved identifier
DUMP	reserved identifier
RANK	reserved identifier
! (terminate scan of current card)	11-0
# (force character into string)	12-7-8

Typographical features such as blank space or change to a new line have no significance to the compiler except that blanks may not appear within basic symbols, identifiers and numbers. Otherwise blank spaces and blank lines may be used freely to facilitate reading.

An exclamation mark "!" punched in column one of an Algol source program card will cause the printed listing of the program to begin a new page with that card. In this case the scan of the card continues starting with column two.

II...

ELEMENTS OF THE COMPILER LANGUAGE

CHARACTERS

The Algol compiler employs a character set which is commonly available as a variant of the usual Hollerith code (IBM 026 Fortran H set) together with a few special multipunch 1107 characters. These characters are:

THE ROMAN ALPHABET

A,B,...,Z

THE ARABIC NUMERALS

0,1,...,9

SPECIAL CHARACTERS

\+
\-
=
(
)
.
,
$
/
*
<space>
'
: (a 5-8 punch)
& (a 2-8 punch)

<	(a 12-6-8 punch)
>	(a 6-8 punch)
;	(a 11-6-8 punch)
[	(a 12-5-8 punch)
]	(a 11-5-8 punch)
!	(a 11-0 punch)
#	(a 12-7-8 punch) (later referred to as "pound sign")

In addition, some multiples of characters are given meaning as though they constituted a single character:

**	exponentiation
..	:
&&	base-10 scale factor double precision
//	integer divide $\dot{-}$

From these characters, statements are constructed which are translated by the compiler into machine language for execution by the Univac 1107.

METALINGUISTIC SYMBOLS

In addition to the script letters used in the text, some symbols will be employed with metalinguistic significance. These symbols include:

SYMBOL	SIGNIFICANCE
$\equiv$	is equivalent to has the form of
ρ	relational operator
ω	arithmetic operator
Δ	space

IDENTIFIERS

The fundamental construct of the Algol language is the identifier. Identifiers are used to name the various things that make up a program, such as variables, functions, labels and procedures. An identifier is a

string of letters and digits subject to the following conditions:

1) The first character must be a letter.
2) No special characters (including spaces) may be embedded within an identifier (the only exception is GO TO).
3) Any number of characters may be used. However, the compiler considers two identifiers to be the same if their first twelve characters are the same (six for external system names, eleven for procedures used in the functional sense, three for defined variables).

A few identifiers are reserved by the compiler for use as operators and punctuation marks. These reserved identifiers should not be used by the programmer in any context other than that set down in this manual.

Any other identifiers may be used at the programmer's discretion. However, some other words, called predefined identifiers, are known by the compiler as the names of library functions or intrinsic functions and should be used with caution if the programmer wishes to employ the standard library element in his program.

A list of reserved identifiers and predefined identifiers is given in Appendix I. Whenever they occur in examples in this manual they are underlined.

EXAMPLES:

Z
BEGIN
PARKAVENUESOUTH
ENTIER
A374
U1107POINT5
COMMENT
RUNGEKUTTAGILL
GTR

QUANTITIES

The compiler is concerned with the manipulation of six types of

quantities: real quantities, integer quantities, Boolean quantities, double precision quantities, complex quantities, and string quantities.

Real Quantities represent the class of real numbers to an accuracy of eight significant decimal digits, the maximum permitted by the word length of the Univac 1107.

Integer quantities represent the class of integers that can be expressed in the word length of the Univac 1107, i.e. the integers whose magnitude is less than $2^{35}-1$.

Double precision quantities represent the class of real numbers with a precision of sixteen significant decimal digits.

The magnitude of a real quantity (single or double precision) must be less than 10^{38}. Any real quantity which is less than 10^{-38} in magnitude, is represented by zero.

Boolean quantities represent truth values. The only values for Boolean quantities are TRUE and FALSE.

Complex quantities represent the class of complex numbers, which are expressed as an ordered pair of real numbers.

String quantities represent the class of strings of valid characters. A maximum of 4095 characters is permitted in any single string.

A program may contain quantities of any or all of these types. The programmer assigns the types of the variables and evaluated procedures that appear in his program.

VARIABLES

Variables treated by this compiler are two kinds--simple variables and variables with subscript(s). A simple variable represents a single quantity and is denoted by an identifier; a variable with subscript(s) represents either 1) a single element of an array which is denoted by the identifier which names the array, followed by a subscript list enclosed in parentheses, or 2) a portion of a string variable. A subscript list consists of arithmetic expressions separated by commas.

EXAMPLES:

Simple Variables:

X

ALPHA

C13

Variables with Subscripts:

```
A(I,J)
M(I + 1, J + 1)
V(F(P + 1), 12 + Q)
Q(W(T), X(T), Y(T), Z(T))
C(13)
```

The expressions (see Chapter III) which make up the subscripts of a variable with subscripts may be of any complexity. Real values are allowed, in which case the real number is rounded to the nearest integer (see the description of the INTEGER function in Chapter VIII). Each subscript expression must have a value which is not less than the minimum and not greater than the maximum specified for that array by the ARRAY declaration or for the string as specified by the STRING declaration (see Chapter V). The number of subscript expressions must equal the number of dimensions of the array as declared in the ARRAY declaration for that array, or be less than three in the case of string variables.

A string variable may have zero, one or two subscripts. For example, if S is a string variable then

```
S(I,J)
```

refers to the substring of J characters taken from S in ascending order starting with the character in the Ith position.

The second subscript may be omitted, in which case it is assumed to have a value of one. Therefore

```
S(I)
```

refers to the single character in the Ith position of the string S. If both subscripts are omitted then it is understood that the entire string is being referenced. (See Chapter VII, STRINGS.)

The "declaration of type" described in Chapter V determines whether a variable represents an INTEGER, REAL, REAL2 (double precision), COMPLEX, STRING or BOOLEAN quantity.

INTEGER CONSTANTS

Integer constants may be represented in either the decimal or octal number systems.

A decimal integer consists of a string of one to ten decimal digits. Leading zeroes are ignored and imbedded spaces are not permitted.

EXAMPLES:

0
15
16384
2121

If the "K" option is used on the Algol control card the compiler interprets an integer constant with one or more leading zeroes as an octal number. The magnitude of an octal constant must be less than 8^{12}. Only the digits 0, 1, ...,7 may be used in an octal constant. When the compiler goes into octal mode the word OCTAL is printed at the left of the listing. The "K" option may be invoked selectively in a program by means of trace number 25. See Chapter XIII for a description of option letters and trace numbers.

EXAMPLES:

0
017
040000
04111

REAL CONSTANTS

Real constants are represented by a string of digits which contains "." -- a decimal point. The decimal point may not appear at the end of the string. A real constant may contain a maximum of eight digits, significant or not.

EXAMPLES:

3.1415927
43.0
.6394

If desired, a scale factor may be appended to a real constant to indicate that it is to be multiplied by the indicated power of 10. This scale factor is written as an ampersand (&) followed perhaps by a + or - sign and then by an integer. The integer specifies the power of 10 to be used, and is limited to a two-digit number.

EXAMPLES:

2.6&5 means 2.6×10^{5} or 260,000.0
1.7&-3 means 1.7×10^{-3} or 0.0017

A third option allows a real number to be written as an integer followed by a scale factor.

EXAMPLE:

3&+4

This is precisely equivalent to writing 3.0&+4 or 30,000.0. Note that a scale factor alone may be used to specify a real number--for example 10^2 may be written as &+2, etc.

REAL2 CONSTANTS (DOUBLE PRECISION)

Double precision constants are represented by a string of more than eight digits which contains "." -- a decimal point. A double precision constant may contain a maximum of sixteen digits.

EXAMPLES:

0.00006174205

2.71828182845904

If desired, a scale factor may be appended to a double precision or real constant to indicate that it is to be multiplied by the indicated power of 10. This scale factor is written as two ampersands followed perhaps by + or - sign and then by an integer as in the real case.

EXAMPLE:

1.0&&-1 or &&-1

This represents 0.1 correct to sixteen significant figures. One should note that 0.1 is not represented exactly in binary, and that double precision representation will be more accurate than the normal single precision.

NOTE: The ampersand (&) is used only in writing constants. Exponentiation of variables in a program is denoted by "**".

COMPLEX CONSTANTS

Complex constants are represented as an ordered pair of real constants separated by a comma and enclosed by corner brackets < >.

EXAMPLES:

<1.0, 1.0> represents 1 + i

<-3.4, -1.0 & -2> represents -3.4 + 0.01i

BOOLEAN CONSTANTS

Only two Boolean constants are allowed: TRUE and FALSE.

STRING CONSTANTS

String constants are represented by any string of acceptable characters (excluding an apostrophe, exclamation mark, or pound sign) enclosed by apostrophes. The exclamation mark (!) terminates the string on the current card and continues it with the first non-blank character on the next card. If a string constant is continued from one card to the next without being terminated by an exclamation point on the earlier card, then the string resumes with column one of the next card. The pound sign (#) forces the next character on the card into the string no matter what it is, as in the following examples:

Characters punched on card	Effective string
'HOW△NOW△BROWN△COW'	HOW NOW BROWN COW
'128F6.2'	128F6.2
'IT△AIN'T△NO△USE'	IT AIN'T NO USE
'WHEW#!#!'	WHEW!!
'KEY△OF△F##'	KEY OF F#

CAUTION: The only constant that may contain an embedded space (blank) is a constant of type STRING.

EVALUATED PROCEDURES

The compiler allows the use of a wide variety of functions. In this section we will consider only the simplest form of functional notation in order to provide a basis for the next chapter. (Chapter XII contains a complete description of the use of procedures and the manner in which they are defined.) For the moment we will assume that a procedure acts on one or more quantities called arguments and produces a single number as a result. This resulting quantity is called an evaluated procedure.

GENERAL FORM:

<identifier> ($\langle exp_1 \rangle$, ..., $\langle exp_N \rangle$)

where <identifier> is the name of the procedure and $\langle exp_1 \rangle$ through $\langle exp_N \rangle$ are expressions which are the arguments of the procedure.

EXAMPLES:

```
SIN(X)
SQRT(B**2-4*A*C)
HYPERGEOM(A,B,C,Z)
LN (SIN(THETA-ALPHA/2))
```

The type of a procedure depends on the manner in which the procedure was defined. The type required for each of the arguments is also determined by the definition of the procedure. It is the programmer's responsibility to ensure that each of the arguments is of the proper type. However, VALUE parameters will have the arithmetic converted if possible, and all other violations will cause an error message.

III...

EXPRESSIONS

Algol statements deal with three kinds of expressions: Arithmetic expressions (those having numerical values), Boolean expressions (those having truth values), and Designational expressions (those having statement labels as values). This chapter describes the manner in which these expressions may be combined to produce new expressions. Expressions must be well formed in accordance with mathematical convention and with the rules set forth below.

ARITHMETIC EXPRESSIONS

Arithmetic quantities are combined by means of the operations +, -, *, /, //, and **. The symbol ** is used to denote exponentiation, (↑), [i.e. B**2 has the meaning of B^2 or (B↑2)], and // denotes integer divide (÷). In addition to these six symbols, parentheses may be employed to indicate that a specific order of evaluation is to be followed rather than the assumed order in ALGOL. To be explicit, it is assumed -- in the absence of parentheses to indicate otherwise -- that exponentiation is performed before multiplication and division, multiplication and division before addition and subtraction. An option is provided for assigning a higher precedence to multiplication (*) than division (/) or (//). Operations on the same level (e.g. addition and subtraction) are done from left to right. Parentheses should be used to express the exact meaning desired.

A variable, a constant, or an evaluated procedure of type INTEGER, REAL, REAL2, COMPLEX, or STRING will in itself constitute an arithmetic expression. Furthermore, if

<arith exp$_1$>, <arith exp$_2$>

are arithmetic expressions and

<Bool exp>

is a Boolean expression, then each of the following is also an arithmetic expression:

<arith exp$_1$> * <arith exp$_2$>	<arith exp$_1$> + <arith exp$_2$>
<arith exp$_1$> / <arith exp$_2$>	<arith exp$_1$> - <arith exp$_2$>
<arith exp$_1$> ** <arith exp$_2$>	+ <arith exp$_1$>
(<arith exp$_1$>)	- <arith exp$_1$>
<arith exp$_1$> // <arith exp$_2$>	IF <Bool exp> THEN <arith exp$_1$> ELSE <arith exp$_2$>

The arithmetic operation denoted by a double slash (//) is called "integer division". If A and B are of type INTEGER and B $\neq$ 0, then the result of A//B is equal to the value of the expression

SIGN(A/B) * ENTIER(ABS(A/B))

See Chapter VIII for descriptions of the SIGN, ENTIER and ABS functions.

EXAMPLES:

Expression	Compiler interpretation
A+B*C	A+(B*C)
A*B+C	(A*B)+C
A*B/C	(A*B)/C
-X*Y	(-X)*Y
-X-Y	(-X)-Y

NOTE: Exponentiation of expressions cannot be accomplished by use of the ampersand (&). The expressions

<arith exp$_1$> & <arith exp$_2$>

<arith exp$_1$> && <arith exp$_2$>

are meaningful only under the conditions described in Chapter II, Real Constants and Real2 Constants.

STRINGS IN ARITHMETIC EXPRESSIONS

Whenever a string quantity (variable, constant, or procedure) is used within the context of an arithmetic expression, then the string is assumed to be a string of digits and is converted automatically to an integer quantity denoting the value of the string. If the assumption of a string of digits is false an error at run time will be detected and a message "Improper string conversion" will be indicated.

As an example, let:

A be the string '4'

B be the string '9'

then A*B has the value 36

and A**'2' has the value 16.

Note: '2' + '36' is equivalent to 2+36 as an arithmetic expression.

BOOLEAN QUANTITIES

Boolean quantities may be combined by means of logical operations to form Boolean expressions in a manner entirely analogous to the combination of arithmetic quantities by arithmetic operations. Boolean expressions are again true or false, depending entirely on the truth values of the quantities entering into the expression and the definitions of the Boolean operations combining them.

RELATIONAL OPERATORS

Another class of Boolean expressions is comprised of those which result from a test on arithmetic expressions. These are termed arithmetic relations, and consist of two arithmetic expressions and a relational operator. The latter is an operator in the sense that it performs a transformation on the comparison to produce a truth value. This value is either true or false depending upon the results of the comparison.

GENERAL FORM:

<arith exp$_1$> <rel oper> <arith exp$_2$>

where <arith exp$_1$> and <arith exp$_2$> are arithmetic expressions and <rel oper> is a relational operator. The relation has the value TRUE if <arith exp$_1$> is in the relation to <arith exp$_2$>; otherwise its value is FALSE.

The relational operators employed in the Algol compiler are GTR, GEQ, EQL, LEQ, LSS and NEQ. Their significance is indicated in the following table:

EXPRESSION	CONVENTIONAL MATHEMATICAL NOTATION	MEANING
<arith exp$_1$> GTR <arith exp$_2$>	$A_1 > A_2$	greater than
<arith exp$_1$> GEQ <arith exp$_2$>	$A_1 \geq A_2$	greater than or equal t
<arith exp$_1$> EQL <arith exp$_2$>	$A_1 = A_2$	equal to
<arith exp$_1$> LEQ <arith exp$_2$>	$A_1 \leq A_2$	less than or equal to
<arith exp$_1$> LSS <arith exp$_2$>	$A_1 < A_2$	less than
<arith exp$_1$> NEQ <arith exp$_2$>	$A_1 \neq A_2$	not equal to

Spaces are required to the left and right of the relational operator.

EXAMPLES:

```
X NEQ 0
ABS(L - LPRIME) LSS EPSILON
T GTR TMAX
```

STRING IN RELATIONS

Strings may be compared by the above relational operators. The comparisons are made using the natural collating sequence of characters on the 1107 (i.e. Fieldata code). If a non-standard collating sequence is desired

then a declaration of RANK may be made (See Chapter VII).

A string comparison is made only if both sides of the relation are strings. If either operand is non-string then the string operand is converted automatically to its associated arithmetic expression before performing the comparison as in the arithmetic expression case.

BOOLEAN OPERATIONS

The Boolean operations which are accepted by the compiler are NOT, AND, OR, XOR, IMPL, and EQUIV. These operations are called negation, conjunction, disjunction, exclusive disjunction, implication, and equivalence, and are defined as follows (P and Q are Boolean quantities):

P	Q	NOT P	P AND Q	P OR Q	P XOR Q	P IMPL Q	P EQUIV Q
false	false	true	false	false	false	true	true
true	true	false	true	true	false	true	true
true	false	false	false	true	true	false	false
false	true	true	false	true	true	true	false

CONSTRUCTION OF BOOLEAN EXPRESSIONS

Any variable, constant, evaluated function, or procedure will itself constitute a Boolean expression, if it is of Boolean type.

In addition, given the arithmetic relation

<arith rel>

and the Boolean expressions

<Bool exp_1>, <Bool exp_2>, <Bool exp_3>

then each of the following is also a Boolean expression:

(<arith rel>)

(<Bool exp_1>)

<Bool exp_1> OR <Bool exp_2>

<Bool exp_1> AND <Bool exp_2>

<Bool exp_1> EQUIV <Bool exp_2>

<Bool exp_1> XOR <Bool exp_2>

<Bool exp_1> IMPL <Bool exp_2>

NOT <Bool exp_1>

IF <Bool exp> THEN <Bool exp_1> ELSE <Bool exp_2>

PRECEDENCE OF BOOLEAN OPERATIONS

Conventions for the order of precedence of Boolean operations are not as well established as are those for arithmetic operations. However, we shall assume the following order, which is apparently the most common:

Unless indicated otherwise by the use of parentheses, NOT will be executed before AND; AND will be executed before OR and XOR; OR and XOR will be executed before IMPL; and IMPL will be executed before EQUIV. In the case of equal priorities, operations are executed from left to right. For example, P IMPL Q IMPL R means ((P IMPL Q) IMPL R).

EXAMPLES:

NOT(P AND Q) OR R IMPL P OR NOT Q

NOT (NOT P) EQUIV P

P IMPL P OR U AND V

(P OR Q) AND NOT (P AND Q)

(A LEQ X) AND X LEQ B

(ERROR LSS TOLERANCE) XOR (N GTR 40)

R AND S OR (F(X) EQL 4)

(M*N*(R-2) + 4 LSS TAN(BETA-ALPHA)) OR FLAG

(U*SINH(M) GTR M7) EQUIV (V*COSH(M) GTR M12)

Boolean expressions may only be used in other Boolean expressions.

DESIGNATIONAL EXPRESSIONS

Designational expressions are those expressions whose values are

statement labels. The form of a designational expression is either a label, a switch variable or a conditional expression between designational expressions. For example, each of the following is a designational expression:

<label>

<switch> (<arith exp>)

IF <Bool exp> THEN <desig exp_1> ELSE <desig exp_2>

ARITHMETIC EXPRESSIONS AND BOOLEAN RELATIONS

Entries in tables represent the arithmetic mode in which calculation or comparison is carried out. The following abbreviations are used.

R	Real
S	String
I	Integer
R2	Double Precision Real
C	Complex
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation
Rel	Relational Operators (GTR, GEQ, EQL, LEQ, LSS, NEQ)

TABLE 1

Type of Right Operand

Type of Left Operand	+ - * /	S	I	R	R2	C
	S	I §	I §	R	R2	C
	I	I §	I §	R	R2	C
	R	R	R	R	R2	C
	R2	R2	R2	R2	R2	C
	C	C	C	C	C	C

§ In these cases division (/) is done in Real mode.

Integer division (//) is always done in Integer mode. If either operand is complex, integer division is undefined.

TABLE 2

Type of Right Operand

Type of Left Operand	Rel	S	I	R	R2	C
	S	S	I	R	R2	C §
	I	I	I	R	R2	C §
	R	R	R	R	R2	C §
	R2	R2	R2	R2	R2	C §
	C	C §	C §	C §	C §	C §

§ If either operand is complex, only EQL and NEQ are valid relational operators.

Entries in Table 3 denote the type of the result. The method of computing the result of exponentiation depends on the values of the operands.

TABLE 3

Type of Exponent

Type of Left Operand	**	S	I	R	R2	C
	S	I	I	R	R2	C
	I	I	I	R	R2	C
	R	R	R	R	R2	C
	R2	R2	R2	R2	R2	C
	C	C	C	C	C	C

IV...

STATEMENTS

The statement is the fundamental unit of expression in the description of an algorithm. Most of what follows in this manual deals with the formation of statements and their interrelation to form larger constructs. Statements may be divided into two classes -- the operational statement and the declarative statement. Operational statements specify something that the object program is to do. Declarative statements give information to the compiler about the program being compiled. After this chapter, the word statement will usually be employed to mean an operational statement; a declarative statement will then be called a declaration. However, for the present, "statement" will stand for either sort.

The first part of this chapter discusses one particular kind of operational statement -- the assignment statement. The last part of the chapter deals with the grammar of statements in general, using assignment statements for examples.

THE ASSIGNMENT STATEMENT

The assignment statement specifies an expression which is to be evaluated and a variable which is to have the resulting value assigned to it.

GENERAL FORM:

<variable> = <exp>

Note that the symbol " = " is used in a special sense in this compiler to signify the process of assignment or substitution of values. Thus X = X + 1 means "using the current value of the variable X, evaluate the expression X + 1, and assign the result as the new value of X". Although X = X + 1 is

not a valid equation, it is a well-formed operational statement, and the compiler will carry out the indicated substitution. Thus the following valid algebraic expression:

```
X**2 = Y + 2
```

has no meaning to the compiler, while

```
X = SQRT(K)
```

is a valid statement, and can be executed by a compiled program, which will assign the value of $K^{\frac{1}{2}}$ to the variable X.

ARITHMETIC ASSIGNMENT STATEMENTS

If the variable in

<variable> = <arith exp>

is of type INTEGER, REAL, REAL2, or COMPLEX then we have an arithmetic assignment statement. If <variable> and <arith exp> are of different type then one of two cases applies:

1) <arith exp> will be converted to the type of <variable> before the assignment is made;
2) the compiler will indicate that the conversion of type is undefined and machine code will not be generated for the statement in question.

EXAMPLES:

```
R = (-B + SQRT(B**2 - 4*A*C))/(2*A)
U = X*COS(X*COS(THETA) + Y*SIN(THETA))
OMEGA = 1/SQRT(L*C)
E = M*C**2
C(I,J) = C(I,J) + A(I,K)*B(K,J)
```

STRING ASSIGNMENT STATEMENTS

If the variable in

<variable> = <exp>

is a string variable, we have a string assignment statement. In this case

<exp> must be either of type string or an arithmetic expression. In the latter instance, <exp> is first converted to type integer, if possible, and then into the corresponding string of digits.

In all cases the replacement is made such that the left most character of the right hand side replaces the left most character in the left hand string variable. Extra spaces are supplied to the right as necessary to fill out the left hand string and any excess of characters from the right hand side will be dropped (i.e. the replacement is left justified in the left hand string variable).

As an illustration, consider the following uses in which A is a string variable of length six:

A before	Statement	A after
ABCDEF	A='XYZUVW'	XYZUVW
ABCDEF	A='LOOP-DE-LOOP'	LOOP-D
ABCDEF	A='HOW'	HOW
ABCDEF	A(2)='Q'	AQCDEF
ABCDEF	A(2,3)='XYZ'	AXYZEF
ABCDEF	A(2,3)=69	A69 EF
ABCDEF	A=3.1415927	3
ABCDEF	A(2,2) = -7	A-7DEF

EXAMPLES:

```
S=1000*SIN(X)
S=IF X GTR Y THEN 'P1' ELSE 'P2'
S(1, N)=S(2, N-1)
```

One word of caution, the string replacements are performed a character at a time starting with the left most character; hence, a replacement of the form:

```
S(2, N-1) = S(1, N-1)
```

will result in the character in the 1,1 position, S(1, 1), being propagated down the string (i.e. the first N characters of the string S will all be the same as the character in S(1, 1)). In order to shift the characters in a string right, it is necessary to first move the string into another string of the same length. For example, let S and T be strings of the same length, then a right shift of one place can be performed on S by:

```
T = S $
S(2, N-1) = T(1, N-1)
```

This operation leaves the character in S(1) unchanged.

BOOLEAN ASSIGNMENT STATEMENTS

If the variable in

<variable> = <Bool exp>

is a Boolean variable, we have a Boolean assignment statement.
EXAMPLES:

```
FLAG = (SWITCH4 OR SWITCH5) AND FLAGPRIME
TEST = (X NEQ 0) AND (Y NEQ 0)
TOGGLE3 = TOGGLE4 AND TAG OR (U LSS V)
```

GENERALIZED ASSIGNMENT STATEMENT

GENERAL FORM:

$$\langle \text{variable}_1 \rangle = \langle \text{variable}_2 \rangle = \ldots = \langle \text{variable}_N \rangle = \langle \text{exp} \rangle$$

If it is desired to assign the same value to a number of variables, it can be accomplished in a single statement by employing this generalized form.

Note that if the list of variables to which a value is being assigned is mixed type, then conversion of type will be performed; e.g., assume X, Y, <exp> are real, and I is integer. Then the statement

X = I = Y = <exp>

will cause the value of <exp> to be stored into Y, rounded to an integer before storing into I, and then this rounded result will be converted to type REAL and stored into X. Thus, for the above condition, X = I = Y = <exp>, X = Y = I = <exp>, and I = X = Y = <exp> may all give different results when <exp> is real. The result of the rounding process is described under INTEGER function in Chapter VIII.
EXAMPLES:

```
V = X = Y = 15.302
A(I) = B(I) = Z = 0
```

THE GRAMMAR OF STATEMENTS

This section discusses certain definitions and rules of the compiler language which have to do with the writing of statements. The basic rule of the grammar of statements is that a statement must be separated from a following statement by a dollar sign (or semicolon).

Even though a statement ends on a given line and the next statement begins on the next line, the separating dollar sign must be indicated. The end of a line has no meaning as punctuation.

GENERAL FORM:

$$\langle \text{statement}_1 \rangle \ \$ \ \langle \text{statement}_2 \rangle \ \$ \ \ldots \ \$ \ \langle \text{statement}_N \rangle$$

Unless otherwise indicated, statements are performed one after the other in the sequence in which they are written. As many statements as desired may be written on a line (subject of course to the physical limitations of the input medium), or a statement may use as many lines as are required for its expression.

EXAMPLE:

```
W = A + B $ X = A - B $ Y = A*B $ Z = A/B
```

COMPOUND STATEMENTS

It is frequently desirable to group several statements together to form a larger construct which is to be considered as a single statement. Such a construct is called a compound statement.

GENERAL FORM:

BEGIN $\langle \text{statement}_1 \rangle$ \$ ... \$ $\langle \text{statement}_N \rangle$ END

The words BEGIN and END serve as opening and closing statement parentheses.

Throughout this description of the compiler, unless the contrary is specifically stated, the word "statement" should be construed to mean either a simple or a compound statement.

Certain other constructs involving the grouping of several statements

automatically constitute compound statements. These will be discussed further in their proper context in CHAPTER XI.

EXAMPLES:

```
BEGIN  U = -B/(2*A) $ V = SQRT(U**2 - C/A) $
R1 = U + V $ R2 = U - V  END
BEGIN  S = SIN(THETA) $  C = COS(THETA) $
X1 = C*X + S*Y $  ETA = -S*X + C*Y  END
```

STATEMENT LABELS

It is often necessary to attach a name to a statement (e.g. if one wishes to get from the end of a program back up to the beginning). This name is called a statement label. A statement label must be an identifier.

GENERAL FORM:

<identifier> .. <statement>

EXAMPLES:

```
START.. SUM = 0
LEGENDRE..  P(N) = ((2*N - 1)*P(N - 1) - (N - 1)*(N - 2))/
ROTATE.. BEGIN  S = SIN(THETA) $ C = COS(THETA) $
         X1 = C*X + S*Y $ ETA = -S*Y + C*Y  END
```

When using a compound statement, the programmer may insert a letter strin after the word END. The comment is terminated by the next "$", END, or ELSE. This may be done for readability of the print-out produced during compilation; the compiler itself makes no use of the information.

GENERAL FORM:

<label> .. BEGIN <statement$_1$> $ ... $ <statement$_N$> END <sequence of symbols not containing "$", END or ELSE>

EXAMPLE:

```
ROOTS.. BEGIN U = -B/(2*A) $ V = SQRT(B**2 - 4*A*C)/(2*A) $
R1 = U + V $  R2 = U - V  END ROOT PROCESSING SECTION  $
```

V ...

BASIC DECLARATIONS

The declarations of type -- INTEGER, REAL, REAL2, COMPLEX, STRING, BOOLEAN -- are explained in this chapter, together with the ARRAY, COMMENT, OWN, DEFINE, SWITCH and LOCAL declarations. These do not exhaust the entire set of declarations available to the programmer; the others are found in later chapters, where they may be treated at greater length.

Declarations determine how the compiled program will handle certain of its elements. It is always necessary to precede the use of an identifier with a declaration of its type.

Upon declaration the value assigned to each variable is "undefined" unless the variables are OWN, in which case all are set to zero except Booleans, which are set to FALSE and strings, which are set to blanks.

DECLARATIONS OF TYPE

Declarations of type are used to indicate that a specified set of identifiers represent quantities of a given type.

GENERAL FORM:

INTEGER	<type list>	$
REAL	<type list>	$
REAL2	<type list>	$
COMPLEX	<type list>	$
STRING	<type list>	$
BOOLEAN	<type list>	$

These statements declare the identifiers given in <type list> to be of integer, single precision real, double precision real, complex, string and Boolean respectively. Note the integer "2" in the declaration of double

precision quantities. A <type list> consists of a sequence of identifiers separated by commas.

EXAMPLES:

INTEGER GAMMA, Q202, IOU5, VITAE $

BOOLEAN AB6, PARITY, UFO48, AREWE $

THE ARRAY DECLARATION

The ARRAY declaration provides a means of referring to a collection of quantities by the use of a single identifier, and at the same time specifies to the compiler the structure which is to be imposed on this collection. The number of dimensions of an array must be less than 64.

If the identifier of an array requires a declaration of type, then that declaration must immediately precede the word ARRAY in the array declaration (cf. GENERAL FORM).

CONSTRUCTION OF ARRAY DECLARATIONS

The identifier of an array must be described by an ARRAY declaration prior to the use of that identifier in a statement.

GENERAL FORM:

< type > ARRAY < array element >,... , < array element >

where

< type > is any of the possible type declarations (REAL, REAL2, INTEGER, COMPLEX or BOOLEAN). If the type declaration is omitted then REAL will be assumed. The < array element > 's are list items of the array declarator list. These list items take on the form:

<array name> (<lower bound$_1$> : <upper bound$_1$>, ..., <lower bound$_N$> : <upper bound$_N$>) where <lower bound$_J$> and <upper bound$_J$> are arithmetic expressions that specify the lower and upper limits on the Jth dimension, respectively. If the specifications of size are omitted after the name then that array will be assumed to have the same specifications as the next <array element>.

The length of the Jth dimension is given by

INTEGER (<upper bound$_J$>) - INTEGER (<lower bound>) + 1

where INTEGER is the transfer function described in Chapter VIII.

INTEGER ARRAY M, N(1:3, 1:4), CHAR (-1:4, 0:5, 1:6), VECTOR (I:J) $

This declaration reserves twelve cells in storage for each of the two-dimensional arrays M and N, 216 cells for the three-dimensional array CHAR, and J-I+1 cells for the one-dimensional array VECTOR (where J and I are integer variables).

THE STRING DECLARATION

The STRING declaration provides a means of referring to a collection of alphanumeric characters in fieldata code by the use of a single identifier, and at the same time specifies to the compiler the structure which is to be imposed on this collection. It is also possible to subdivide this collection into pieces, each of which may be assigned a different identifier. In effect, this means that string variables may be partitioned into a set of substrings, each with a different identifier. This partitioning may be nested to any depth; however, the partitioning at any level must consist only of disjoint pieces.

CONSTRUCTION OF STRING DECLARATIONS

A string variable must be described by a STRING declaration before the string variable is used.

GENERAL FORM:

STRING <string element>, ..., <string element> $

The <string element>'s are list items of the string declarator list. These list items take on the form:

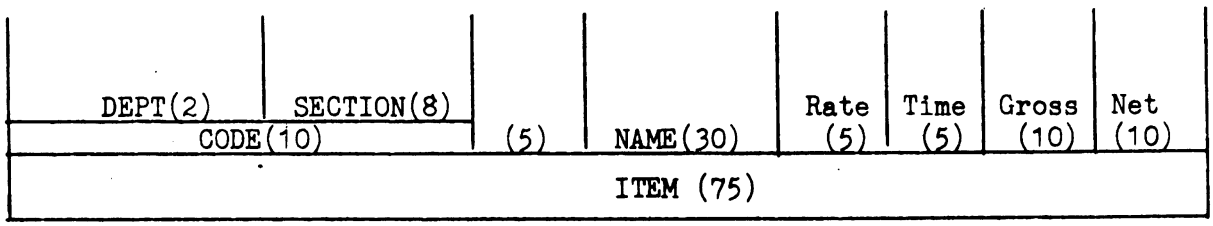
<string name> (<subelement$_1$>, ..., <subelement$_N$>)

where <string name> is an identifier and <subelement$_J$> is either an arithmetic expression which denotes the length of the Jth substring or is itself a <string element>. The length of the entire string (i.e., the total number of characters) is equal to the sum of the lengths of its subelements.

EXAMPLE:

```
STRING CARD(80), LINE(132), ITEM(CODE(DEPT(2), SECTION(8)), 5,
     NAME(30), RATE(5), TIME(5), GROSS(10), NET(10)) $
```

The string CARD holds 80 characters corresponding to a card image. Correspondingly, the string LINE holds one 1004 line image. The string ITEM, on the other hand, has the somewhat complicated structure shown below:



ITEM has 75 characters partitioned into the strings CODE, NAME, RATE, TIME, GROSS, and NET. Also the string CODE of 10 characters is partitioned into the strings DEPT and SECTION.

THE STRING ARRAY DECLARATION

When defining a string array, the general form of the array declaration must be modified to include the length of the string portion along with the partitioning of the string array into substring arrays.

GENERAL FORM:

STRING ARRAY <string array element>, ..., <string array element> $

A <string array element> has the form

<string array name> (<subelement$_1$>, ..., <subelement$_N$> :
<lower bound$_1$> : <upper bound$_1$>, ..., <lower bound$_N$> : <upper bound$_N$>)

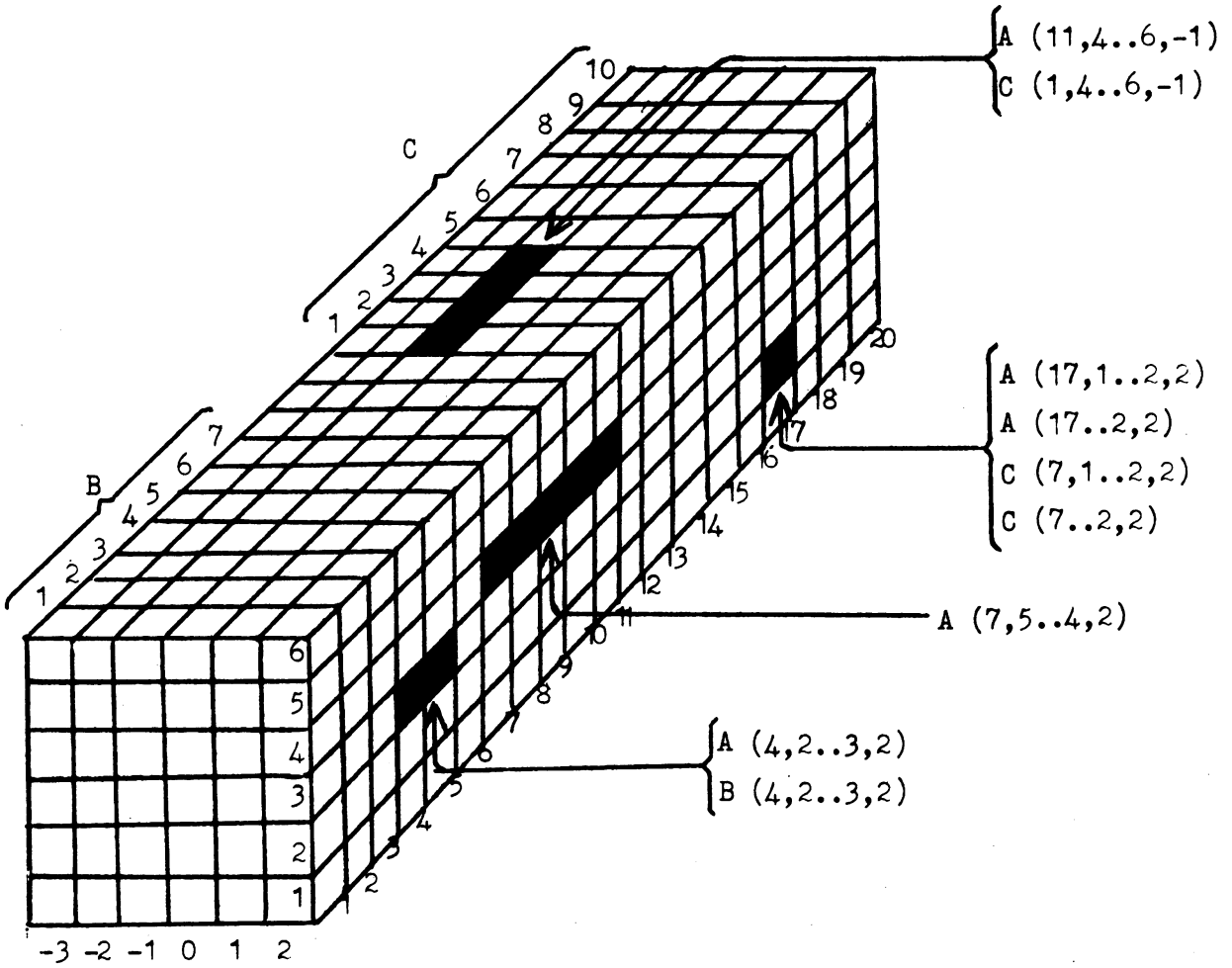
where <string array name> is an identifier, <subelement$_J$> is as described under "THE STRING DECLARATION" and <lower bound$_J$> and <upper bound$_J$> are as described under "CONSTRUCTION OF ARRAY DECLARATIONS". The number of dimensions and their subscript limits apply to each <subelement> as well as to the entire array, i.e., each subelement is itself a string array of N dimensions.

EXAMPLE:

STRING ARRAY A(B(7),3,C(10)..1..6,-3..2) $

This declaration defines a string array A, each of whose elements consists of 20 characters. The array A is partitioned into three subarrays, whose elements contain seven (array B), three, and ten characters (array C), respectively. The main array and subarrays are each two-dimensional, the

subscripts of the first dimension being numbered from 1 to 6, and the subscripts of the second dimension from -3 to 2 (see the accompanying figure).



STRING ARRAY A(B(7),3,C(10)..1..6,-3..2) $

7+3+10=20 6-1+1=6 2-(-3)+1=6

6 * 6 = 36

Each cube in the figure corresponds to one character in the string array A.

In the example of calls on the array shown in the figure, note that the information before the colon specifies the starting point and length of a string,

while the information following the colon specifies which of 36 possible strings is desired.

THE OWN DECLARATION

All of the above declarations will assign space dynamically from local variable storage as needed. When the declarations are no longer in force, this space is made available to the program for other uses. If it is desired to assign permanently the space for the variables, then each of the above declarations must be preceded by the word OWN. The declaration then becomes an OWN declaration. (See Chapter XI, BLOCKS, for details on dynamic storage assignment and OWN variables.)

A FORMAT declaration (described in Chapter IX) and a RANK declaration (described in Chapter VII) may likewise be made into OWN declarations so as to permanently assign storage for such elements.

EXAMPLES:

```
OWN INTEGER A,B,C $
OWN STRING S(100) $
OWN ARRAY FACTORIAL (0..30) $
OWN FORMAT (A1,S72,S8) $
```

CAUTION: In Block 1 the following quantities are always assigned to OWN storage but are not automatically externally defined (see Chapter XII, EXTERNAL REFERENCES): simple variables, arrays, formats, ranks, and generalized variables.

THE DEFINE DECLARATION

A generalized classification scheme for variables has been incorporated into Case Algol. This allows the programmer to define the structure of what are known as "generalized variables" and to declare these variables by means of the DEFINE declaration.

Generalized variables are described in Appendix III and an example of a set of routines that use this feature of Algol is found in the Sort/Merge package, described in Appendix IV.

THE SWITCH DECLARATION

The SWITCH declaration provides a means of selecting a label, switch variable or designational expression from a list by means of a subscript expression on a switch identifier. In effect, the switch declaration defines a switch variable which is similar to a one-dimensional array except that the elements are labels, switch variables or designational expressions.

A switch must have been described by a SWITCH declaration prior to the use of any switch variable with subscripts which represents an element of the switch. The range of subscripts is from 1 to N, where N represents the number of elements in the switch. If a subscript expression on a switch variable falls outside the defined range of the switch then the switch operation is ignored.

The form of a SWITCH declaration is as follows:

GENERAL FORM:

SWITCH <identifier> = <switch element$_1$>, ..., <switch element$_N$> $

where <identifier> is the name of the switch and <switch element$_J$> is a designational expression that represents a statement label.

EXAMPLE:

```
SWITCH  S = L1, IF X GTR Y THEN L2 ELSE L3, L4, T(I+6), L5 $
```

If the switch variable S is referenced from a GO TO statement (see chapter VI) as S(J), then the following transfer of control is made depending upon the value of J.

1) if J = 1 then control transfers to L1.
2) if J = 2 then control transfers to either L2 or L3 depending upon X and Y.
3) if J = 3 then control transfers to L4.
4) if J = 4 then control transfers to switch T.
5) if J = 5 then control transfers to L5.
6) if J < 1 or j > 5 then no transfer is executed.

THE LOCAL DECLARATION

Under certain conditions in a program, the necessity arises for using an identifier at a point when its definition may be ambiguous or unknown (e.g., using a label in a GOTO statement before the occurence of the label definition). If the definition will still be unknown when the current block is ended, the use of the identifier is said to be a forward reference. The case of ambiguity of definition may arise if an identifier has different meanings in different blocks (see Chapter XI, BLOCKS). A parameter to a procedure may also require the LOCAL declaration even though, strictly speaking, it does not constitute a forward reference.

In these cases the identifier must be named in a LOCAL declaration before it is used. This declaration, in effect, localizes the identifier to the proper block as well as supplying information to the compiler about the kind of identifier being used.

GENERAL FORM:

LOCAL <identifier type> <identifier$_1$>, ..., <identifier$_N$> $ where <identifier type> is one of the following: LABEL, PROCEDURE, LIST, SWITCH.

In the case of labels, the LOCAL LABEL declaration is needed in the following cases.

1) A GOTO statement refers to a label which will not be defined until after the current block is ended (forward reference).
2) A GOTO statement refers to a label whose identifier is identical to that of another label in a containing block. This applies when the label will be defined later in the current block.
3) A label is used as a parameter in a procedure call before the label has been defined.

Note that in all cases the LOCAL declaration must appear in the block in which the label is defined.

EXAMPLES:

```
LOCAL LABEL L1,L2,A6,BOX $
LOCAL SWITCH NEXT $
```

THE COMMENT

The COMMENT allows the programmer to include any clarifying remarks, identifying symbols, etc., in the printed compilation. The COMMENT does not appear as part of the compiled program, and has no effect on the program; it merely sets apart any string of characters for printing as part of the compilation. Since the comment extends to the next dollar sign, a dollar sign, obviously cannot be used within the string of characters.

GENERAL FORM:

```
COMMENT S $
```

where S is any string of characters not containing a dollar sign.

EXAMPLES:

```
COMMENT $ SMOOTH FIELD DATA AND REDUCE TO STANDARD FORM $
BEGIN    A = 2*A $  COMMENT $  TWO BARRELS ARE NOT
ENOUGH $ END
```

VI...

CONTROL STATEMENTS

This chapter deals with the means of expressing the 'flow of control' of an algorithm which is to be described in Algol. The order of execution of statements is as important to the description of an algorithm as are the statements themselves.

Control Statements are divided into three sub-classes:

Unconditional control statements, which transfer control to other parts of the program (the GO TO statement).

Conditional control statements, which execute statements contingent on given criteria (the IF statement).

Iterative control statements, which execute statements repetitively (the FOR statement).

UNCONDITIONAL CONTROL STATEMENTS

THE GO TO STATEMENT

The GO TO statement provides the ability to transfer control from one part of the compiled program to another.

GENERAL FORM:

GO TO <desig exp>

or

GOTO <desig exp>

The statement with the label specified by <desig exp> will be executed immediately after the GO TO statement.

EXAMPLES:

```
GO TO START
GOTO S(K)
GO TO IF X GTR Y THEN POGO ELSE S(3+I)
```

Note: In the second example, if K is within the range of definition of the switch S then transfer of control is made to the indicated statement of label. However, if the subscript expression on a switch variable is outside the range of definition on a switch reference, then no transfer of control is performed and control resumes with the next statement after the GO TO and in effect the GO TO is ignored.

CONDITIONAL CONTROL STATEMENTS

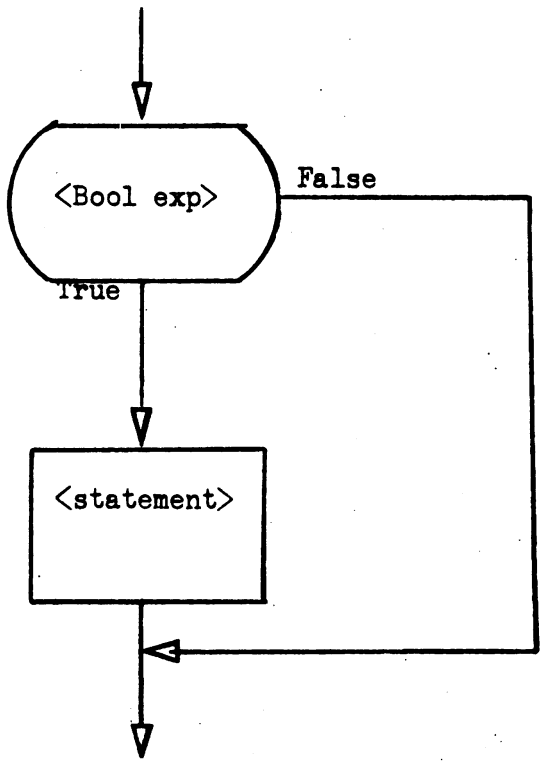
THE IF STATEMENT

The IF statement provides the means of indicating that the next statement in sequence is to be conditionally executed.

FIRST GENERAL FORM:

IF <Bool exp> THEN <statement>

The action of the first form of the IF statement is described graphically by means of the following flow chart:



If the <Bool exp> has the value TRUE then <statement> is executed; if <Bool exp> is FALSE, <statement> is skipped over and control passes to the next statement in sequence.

EXAMPLES:

```
IF X*2 GTR 7 THEN GO TO HOME
IF (I EQL J) THEN A(I,J) = 1
IF (M NEQ 0) OR (N NEQ 0) THEN GO TO LAST
IF (P EQUIV R) OR (P EQUIV S) THEN K = B(J)
IF (X LEQ 0) AND FLAG THEN X = ABS(X)
IF U OR V AND (X LSS 2.4) THEN BEGIN U = 0 $
    V = 0 $ GO TO REPEAT END
```

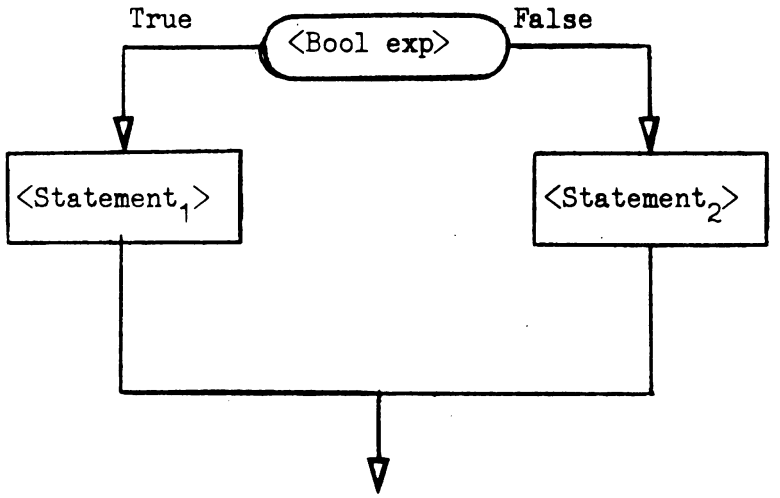
THE ALTERNATIVE FORM OF IF STATEMENT

A second form of the IF statement provides for a choice of alternatives based on the result of a condition.

SECOND GENERAL FORM:

IF <Bool exp> THEN <statement$_1$> ELSE <statement$_2$>

At this point in the program if <Bool exp> is TRUE then <statement$_1$> is performed and <statement$_2$> omitted. If <Bool exp> has the value FALSE then <statement$_1$> is skipped over and <statement$_2$> is executed.



EXAMPLE:

```
INTEGER  R, N $
REAL  PNEXT, PNOW, PPREV, X, NLEGPOLYINX  $
```

```
COMMENT   EVALUATE THE LEGENDRE POLYNOMIAL OF DEGREE R
          IN X USING THE RECURSION RELATION

          (N+1) P(N+1)(x) - (2N+1) X P(N)(X) + N P(N-1)(X) = 0      N = 1, 2, ...

          REFERENCE: FOURIER SERIES AND BOUNDARY VALUE PROBLEMS,
          R. V. CHURCHILL, MCGRAW-HILL, 1941, P.180 $
          IF R LSS 0 THEN
          BEGIN COMMENT YOU MADE AN ERROR $ PNEXT = 1.0&38
          END ELSE
          IF R EQL 0 THEN PNEXT = 1 ELSE
          IF R EQL 1 THEN PNEXT = X ELSE
          BEGIN PNOW = X $ PPREV = 1 $
                FOR N = 2 STEP 1 UNTIL R DO BEGIN
                PNEXT = ((2*N-1)*X*PNOW - (N-1)*PPREV)/N $
                PPREV = PNOW $ PNOW = PNEXT END END $
          NLEGPOLYINX = PNEXT
```

THE FOR STATEMENT

The FOR statement finds its principal use in the control of an iteration where the statement or statement group to be iterated involves a variable (the induction variable) which must take on a succession of values. It is also used to cause a statement to be executed a predetermined number of times.

GENERAL FORM:

FOR <variable> = <for list> DO <statement>

where <variable> may be simple or subscripted, and where <for list> is a sequence of <for list element>'s, which are defined below.

The <for list> describes the sequence of values that <variable> is to assume. For each of these values <statement> will be executed. When the <for list> has been exhausted, the statement following the FOR statement will be executed.

There are three forms of <for list element>'s which can be used to construct a <for list>. The <for list element>'s are separated from each other by commas. A summary of the possible forms is given below:

FIRST FORM:

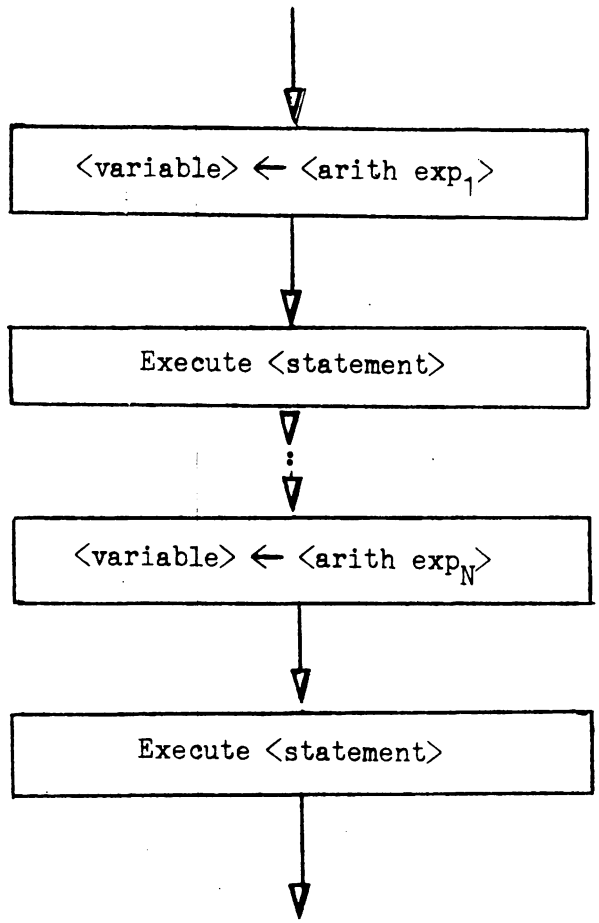
<arith exp>

The value of <arith exp> will be assigned to <variable> and <statement> will be executed before going on to the next <for list element>.

A FOR statement consisting of only these types of elements becomes;

FOR <variable> = <arith exp_1>, ..., <arith exp_N> DO <statement>

which can be interpreted in a flowchart as follows:



That is, <variable> is successively assigned the values of <arith exp_1>, <arith exp_2> and so on through <arith exp_N>. For each value that <variable> assumes, <statement> is executed once.

EXAMPLES:

```
FOR X = 0, Y + 3 * Z, IF Z GTR Y THEN Z ELSE Y, MAX(Z, Y) DO WRITE(X
I = 0 $
FOR A(I) = 2, 3, 5, 7, 11, 13, 17  DO I = I + 1
```

SECOND FORM:

<arith exp$_i$> STEP <arith exp$_d$> UNTIL <arith exp$_f$>

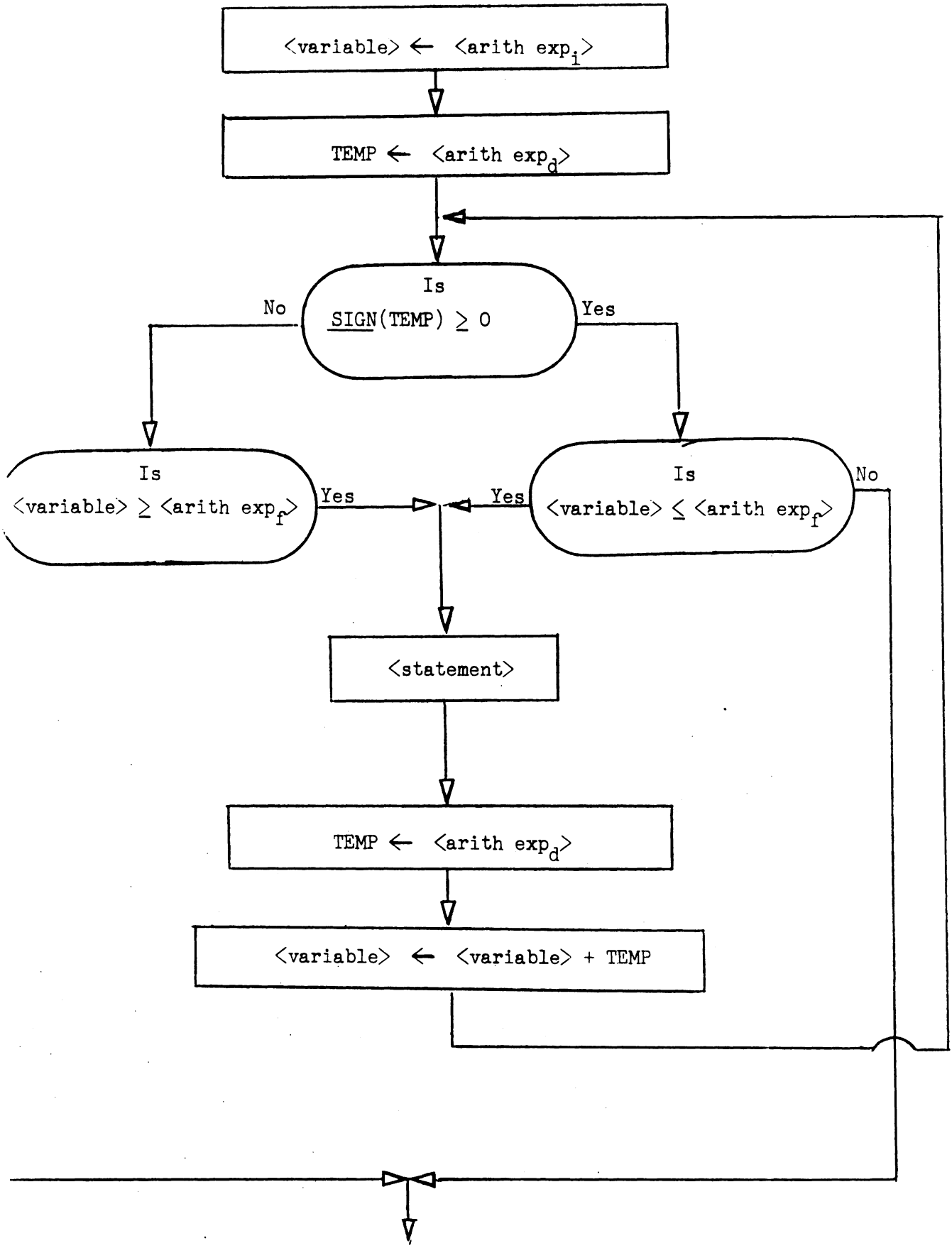
A FOR statement including just one of these elements takes on the form:

FOR <variable> = <arith exp$_i$> STEP <arith exp$_d$> UNTIL <arith exp$_f$>
DO <statement>

This form is equivalent to the simpler statements:

```
          <variable> = <arith exp_i> $
          TEMP = <arith exp_d> $
<label>.. IF (<variable> - <arith exp_f>) * SIGN(TEMP) LEQ 0 THEN
          BEGIN <statement> $
          TEMP = <arith exp_d> $ <variable> = <variable> + TEMP $
          GOTO <label> $ END
```

where TEMP is a variable of the same type as <arith exp$_d$>. This is described by the following flowchart:


<variable> ← <arith exp$_i$>
TEMP ← <arith exp$_d$>
Is
SIGN(TEMP) ≥ 0
No
Yes
Is
<variable> ≥ <arith exp$_f$>
Yes
Is
<variable> ≤ <arith exp$_f$>
Yes
No
<statement>
TEMP ← <arith exp$_d$>
<variable> ← <variable> + TEMP

Note that in any case if the test fails initially, the triplet is considered vacuous and <statement> will not be executed.

If the FOR statement is exited as a result of exhaustion of the <for list>, then the value of <variable>, the induction variable, is considered to be undefined.

EXAMPLES:

```
COMMENT  EVALUATE INNER PRODUCT OF TWO N-VECTORS U AND V $
DOT = 0 $  FOR I = 1 STEP 1 UNTIL N DO
DOT = DOT + U(I)*V(I)

COMMENT  LINEAR EQUATION SOLVER FOR AX = B    WHERE B IS
ADJOINED TO A AND ANSWERS ARE STORED IN B $

FOR J = 2 STEP 1 UNTIL N+1 DO FOR I = 1 STEP 1 UNTIL N DO
BEGIN SUM = 0 $
FOR K = 1 STEP 1 UNTIL MIN(I-1,J-1) DO
        SUM = SUM + A(I,K)*A(K,J) $
A(I,J) = A(I,J) - SUM $
IF I LSS J THEN A(I,J) = A(I,J)/A(I,I)  END $
FOR I = N-1 STEP -1 UNTIL 1 DO
BEGIN  SUM = 0  $
FOR K = N STEP -1 UNTIL I+1 DO
        SUM = SUM + A(I,K)*A(K,N+1) $
A(I,N+1) = A(I,N+1) - SUM  END
```

```
COMMENT MULTIPLY MATRIX A BY MATRIX B
PUTTING THE RESULT IN MATRIX C  $
FOR I = 1 STEP 1 UNTIL N DO
    FOR J = 1 STEP 1 UNTIL N DO
        BEGIN  SUM = 0
            FOR K = (1, 1, N) DO
               SUM = SUM + A(I, K) * B(K, J) $
            C(I, J) = SUM
        END
```

The third and final form that a <for list element> may assume is called a WHILE element.

THIRD FORM:

<arith exp> WHILE <Bool exp>

A FOR statement having a <for list> consisting of one WHILE element has the form:

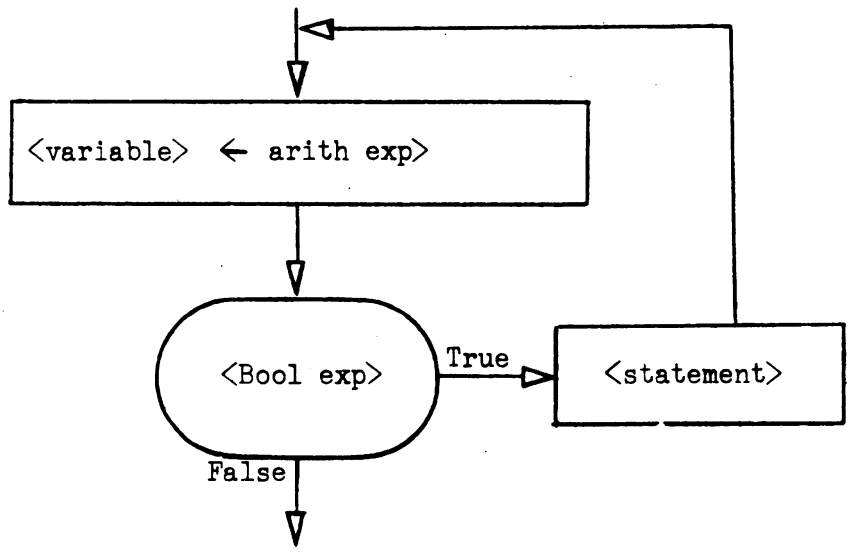
FOR <variable> = <arith exp> WHILE <Bool exp> DO <statement>

The interpretation of this statement is as follows. First <variable> is set equal to <arith exp> and <Bool exp> is evaluated. If the value is TRUE then <statement> will be executed. After the execution of <statement>, <variable> will be assigned the value of <arith exp> and <Bool exp> will again be tested. On the other hand if <Bool exp> is FALSE then <statement> will be skipped and control will resume with the statement following the FOR statement.

In other words this type of FOR statement can be replaced by the following set of statements:

```
<label> ..    <variable> = <arith exp>  $
              IF <Bool exp>  THEN  BEGIN <statement>  $
              GO TO <label>        END
```

which can be expressed by the following flowchart:



If the program exits from the FOR statement because <Bool exp> is FALSE then the value of <variable> is considered to be undefined.

JUMPS IN AND OUT OF FOR STATEMENTS

A GO TO into a FOR statement leads to an undefined operation (i.e., the result is unpredictable). A GO TO out of a FOR statement is acceptable and <variable>, the iterated variable, retains its current value.

It should of course be realized that all three types of for list elements can be combined together in any order to form a general < for list > to be used within a FOR statement. The sequence of values which results is the expected one.

EXAMPLES:

```
FOR Z = 0 STEP 1 UNTIL 10, 15 STEP 5 UNTIL 50,
    100 STEP 50 UNTIL 1000, 5000, 10000 DO
    S = S + SIMPSON (0, Z, &-5, F)
```

This statement would cause the summation to be performed for Z = 0, 1, 2, ..., 9, 10, 15, 20, 25, ..., 50, 100, 150, ..., 950, 1000, 5000 and 10000.

```
FOR I = 1 STEP 1 UNTIL N DO
   FOR J = 1 STEP 1 UNTIL I-1, I+1 STEP 1 UNTIL N DO
   A(I,J) = A(I,J)/A(I,I)
```

This statement would divide off-diagonal elements of each row of the matrix A by the diagonal element of that row. Note that the first <for list element> of the second FOR clause (the one indexed by J) is vacuous when I = 1; the second <for list element> is vacuous when I = N.

VII...

STRINGS

This chapter is a compendium of the basic declarations and operations involving strings. Some of the material found in other chapters is duplicated here in the belief that a unified reference for this subject will prove valuable to the programmer.

STRING QUANTITIES

A string quantity is a sequence of 1107 characters. The number of characters in a string is called its length. The length of a string may be between one and 4095. See Appendix XI for a list of 1107 Fieldata characters and their representations in the computer memory and on I/O devices.

STRING CONSTANTS

A string constant consists of a string of characters (excluding apostrophe, exclamation mark and pound sign) enclosed by apostrophes. The exclamation mark (!) terminates the string on the current card and continues it with the first non-blank character on the next card. The pound sign (#) forces the next character on the card into the string regardless of what it is -- the sequence #' enters the character (') into the string, the sequence #! enters a (!) into the string and the sequence ## erters a (#) into the string.

These considerations do not apply to strings in source program instructions and strings in input data. (See Chapter IX, INPUT/OUTPUT.)

EXAMPLES:

'128F6.2'

'HOW△NOW△BROWN△COW.'

'HELP#!'

where △ denotes a blank (space).

THE STRING DECLARATION

A string variable is used to refer to a sequence of alphanumeric characters that are represented in the 1107 memory by Fieldata code (see Appendix XI). The STRING declaration is used to name the variable, specify its length and indicate, if desired, the partitioning of the string into substrings. Each substring may itself be named and partitioned. The partitioning of a string variable may be nested to any depth.

GENERAL FORM:

STRING <string element>, ..., <string element> $

A <string element> has the form

<string name> (<subelement$_1$>, ..., <subelement$_N$>)

where <string name> is an identifier and <subelement$_I$> is either an arithnetic expression which denotes the length of the Ith substring or is itself a <string element>. The length of the entire string is the sum of the lengths of the subelements.

EXAMPLE:

```
STRING HOBBIT (FRODO(10),BILBO(5),5,SAM(2)) $
```

This declares a string named HOBBIT of 22 characters, which is partitioned into four substrings of 10, 5, 5 and 2 characters.

PREDEFINED IDENTIFIERS IN STRING DECLARATIONS

A predefined identifier may be used in a <string element> in two ways. It may be redefined as the name of a string or substring, as in

STRING CLOUDY (SIN(10),COS(10))

or it may be part of an expression that specifies the string length, as in

STRING WINDY (PART1(+INTEGER(X)),PART2(2*INTEGER(Y)))

where the length of the string WINDY is equal to the value of the expression

INTEGER(X) + 2*INTEGER(Y)

STRING VARIABLES

Any substring of a string variable may be handled by appending subscripts to the identifier of the string. It is not required that the declaration of the string explicitly partition the string into substrings.

Characters of a string are numbered from left to right in ascending order starting at one.

GENERAL FORM:

<string name> (<arith exp$_1$>, <arith exp$_2$>)

denotes the substring of <string name> having length equal to the value of <arith exp$_2$> and starting at the character position specified by <arith exp$_1$>.

EXAMPLE:

QUEUES (I + 5, 2 * K)

denotes the substring of 2*K characters starting at the (I+5)th character of the string QUEUES.

If the second subscript is omitted the form is

<string name> (<arith exp>)

which denotes the single character in the <arith exp> position of <string name>. That is, if the second subscript is absent it is assumed to have a value of one.

If both subscripts are missing, the entire string is referenced.

EXAMPLE:

INVENTORY (X-1)

refers to the character in the (X-1)th position of the string INVENTORY.

CAUTION: The result of an operation involving a string variable whose starting position or length is less than one is undefined.

THE STRING ARRAY DECLARATION

The STRING ARRAY declaration allows a collection of strings to be referred to by a single identifier.

GENERAL FORM:

STRING ARRAY <string array element>, ..., <string array element> $

A <string array element> has the form

<string array name> (<subelement$_1$>, ..., <subelement$_N$> :
<lower bound$_1$> : <upper bound$_1$>, ..., <lower bound$_N$> : <upper bound$_N$>)

where <string array name> is an identifier, <subelement$_I$> is as described under THE STRING DECLARATION, and <lower bound$_I$> and <upper bound$_I$> are arithmetic expressions that specify the lower and upper limits, respectively, on the value of the Ith subscript. The number of dimensions and their subscript limits apply to each subelement identifier as well as to the array identifier, i.e., each subelement is itself a string array of N dimensions. The size of the Ith dimension is given by

INTEGER (<upper bound$_I$>) - INTEGER (<lower bound$_I$>) + 1

NOTE: Although the ARRAY declaration allows dimension specifications to be omitted, the STRING ARRAY declaration requires each <string array element> to include complete information about the lengths and dimensions of the element.

EXAMPLES:

STRING ARRAY STR(12..1..10) $

The array STR consists of ten strings of twelve characters each. The expression

STR(J..L)

refers to the Jth character of the Lth element of the array. The expression

STR(J,K..L)

refers to the K characters starting at the Jth character of the Lth element.

STR(L)

refers to the entire Lth element (i.e., the entire 12-character string).

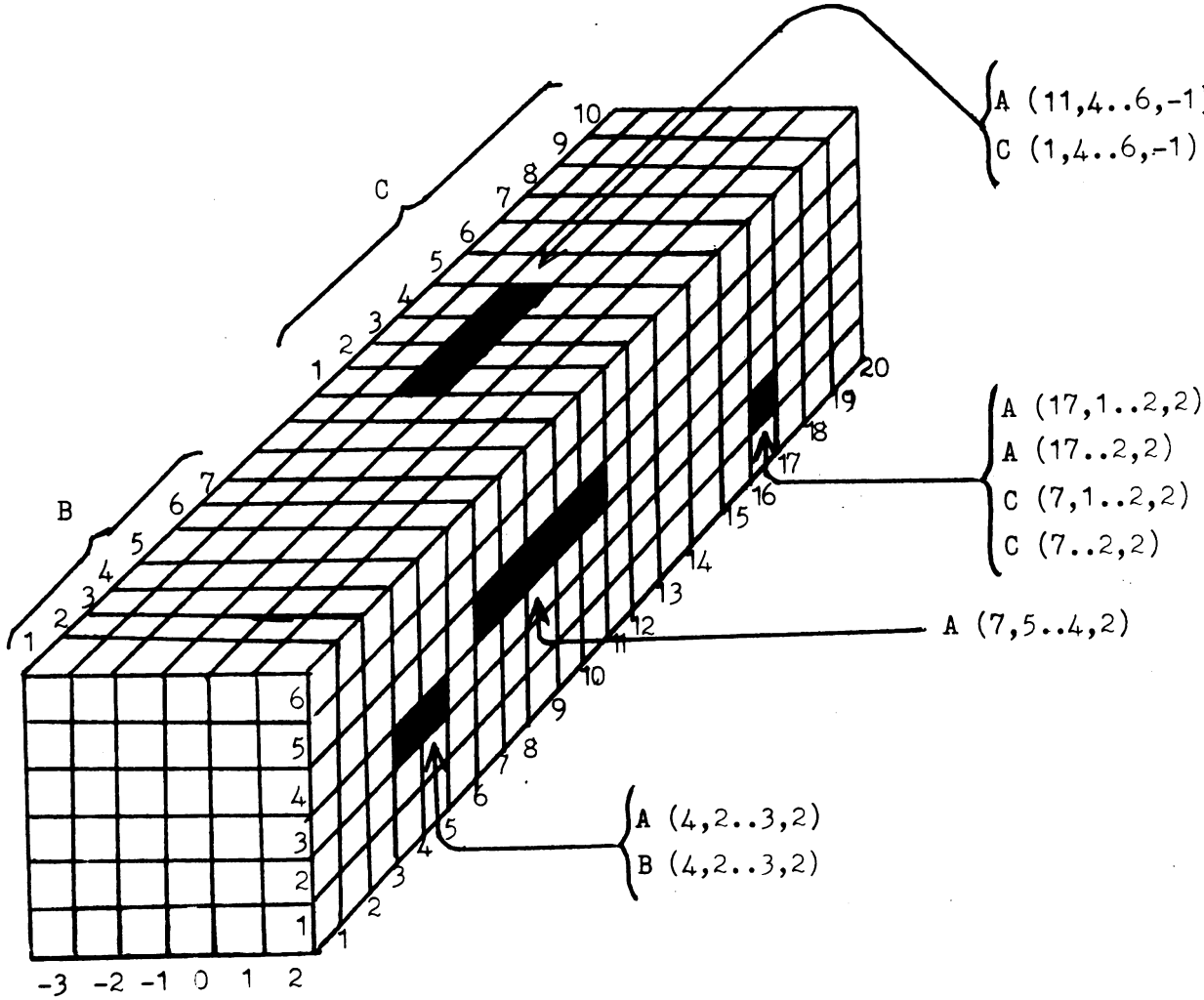
STRING ARRAY STACK1(ORS(6),ANDS(6)..1..N,1..M) $

The array STACK1 is partitioned into two subarrays, ORS and ANDS, and all three arrays are two-dimensional. An element of STACK1, say STACK1(I,J), is partitioned into two substrings of six characters each, namely (ORS(I,J) and ANDS (I,J).

EXAMPLE:

STRING ARRAY A(B(7),3,C(10)..1..6,-3..2) $

This declaration defines a string array A, each of whose elements consists of 20 characters. The array A is partitioned into three subarrays, whose elements contain seven (array B), three, and ten characters (array C), respectively. The main array and subarrays are each two-dimensional, the subscripts of the first dimension being numbered from 1 to 6, and the subscripts of the second dimension from -3 to 2 (see the accompanying figure).



STRING ARRAY A(B(7),3,C(10)..1..6,-3..2) $

7+3+10=20 6-1+1=6 2-(-3)+1=6

6 * 6 = 36

Each cube in the figure corresponds to one character in the string array A.

In the example of calls on the array shown in the figure, note that the information before the colon specifies the starting point and length of a string, while the information following the colon specifies which of 36 possible strings is desired.

STRINGS IN ARITHMETIC EXPRESSIONS

A string quantity may be used as an operand in an arithmetic expression. In this situation an attempt will be made at execution time to handle the string as a string of digits possibly including a sign and spaces. If the attempt succeeds the string will be converted to the corresponding integer value. Otherwise the message

"IMPROPER STRING CONVERSION"

will be printed.

STRING ASSIGNMENT STATEMENTS

GENERAL FORM:

<string variable> = <string exp>

or

<string variable> = <arith exp>

In the second case, <arith exp> is evaluated and converted to type integer and the integer is converted to a string of digits, beginning with a minus sign if the value is negative.

No matter how the righthand string is specified the following rules govern the assignment statement:

(a) the leftmost character of the left side is replaced by the leftmost character of the right side;

(b) if the length of the lefthand string is greater than the length of the righthand string then the lefthand side is space filled in the higher numbered positions;

(c) if the righthand side string length is greater than the lefthand side string length then the excess characters in the higher-numbered

positions are ignored (are not used in the replacement). See Chapter IV for examples.

STRINGS IN RELATIONS

The relational operators GTR, GEQ, EQL, LEQ, LSS and NEQ may be used to compare two string quantities or to compare a string quantity with an arithmetic expression.

GENERAL FORM:

<string or arith exp> <rel oper> <string or arith oper>

If one of the operands is an arithmetic expression and the other is a string, an attempt will be made to convert the string to an integer value and if this is successful the comparison proceeds as in an arithmetic relation.

If both operands are string quantities the operation proceeds by comparing corresponding characters of the two strings starting with the first position. If one string is shorter than the other, the shorter is filled with blanks (octal 05) when necessary to complete the comparison.

Two strings are considered equal if all characters in corresponding positions are of equal rank. If the strings are not equal then the direction of inequality is determined by the leftmost pair of corresponding characters that are not equal.

EXAMPLES:

The following relations are true under the standard collating sequence.

'ABADABA?' LSS 'ABADABA!'

'=>' GTR '<='

THE RANK DECLARATION

All string comparisons are based on the natural sequence as defined by 1107 Fieldata character code (see appendix XI for the usual Fieldata collating sequence). The RANK declaration specifies a string variable which defines a new character collating sequence.

GENERAL FORM:

RANK <rank list element>, ..., <rank list element> $

A <rank list element> defines an ordering relationship among the Fieldata character and have the following form:

<rank name> (<string>)

where <rank name> is an identifier and <string> is of the form

'<$char_1$> <$oper_1$> <$char_2$> <$oper_2$> ... <$char_{N-1}$> <$oper_{N-1}$> <$char_N$>'

where <$char_i$> is any of the 64 Fieldata characters (including space) and <$oper_i$> is any of the four operators =, :, -, <. Note that the ":" in this case cannot be represented by ".." but must correspond to the 5-8 multiple card punch.

The significance of the rank operators is as follows:

$\alpha < \beta$	the character β is assigned a rank one higher than α
$\alpha = \beta$	the character α is assigned to the same rank as β
$\alpha - \beta$	the characters α through β are assigned ranks in ascending order. (α must be less than β in the natural collating sequence.)
$\alpha : \beta$	the characters α through β are assigned to the same rank. (α must be less than β in the natural collating sequence.)

Note that a space (△) is meaningful with a RANK string. It should be noted, that any character not explicitly assigned to a rank will be assigned to zero rank (i.e., below all other assigned characters).

EXAMPLES:

RANK SP ('0:9 < A - Z') $

This declaration generates a table which assigns all numerics to rank 1, the alphabetics A through Z to ranks 2 through 27, and all other characters to rank 0.

THE SETRANK PROCEDURE

The table of values that is generated by a RANK declaration is activated

by the SETRANK procedure. SETRANK may be used to invoke a previously declared <rank list element> or to return to the usual Fieldata collating sequence.

GENERAL FORM:

SETRANK (<rank name>)

This causes the collating sequence defined by the string associated with <rank name> to be put into effect. All string comparisons attempted while it is in effect will make reference to that sequence.

SETRANK

This causes the usual Fieldata collating sequence to be put into effect. In sum, the SETRANK procedure should have one parameter when a new sequence is to be put into effect and no parameter when the usual Fieldata sequence is to be invoked.

The SETRANK procedure is generated in-line by the Algol compiler and its effect is local to the block in which it appears. On exit from a block, considerations of rank that were invoked in that block will no longer apply.

THE RANK PROCEDURE

The numeric value of one or more characters in a string may be obtained by use of the RANK function.

GENERAL FORM:

RANK (<character>)

The result of this function is an integer that represents the rank of <character> according to the collating sequence currently in force.
EXAMPLE:

RANK (NAME(3))

returns the numeric rank of the third character of the string NAME.

Another use of the RANK function has the form

RANK (<string>).

The result of this call on RANK is an integer which falls into one of two classes:

(a) if all characters of the string have the same rank according to the collating sequence currently in effect then the result is that rank;

(b) if all the characters do not have the same rank then the result is the negative of the character position of the leftmost character that does not have the same rank as the character immediately preceding it (note that in this case a result of minus one is never returned). In this case the result returned is negative to distinguish it from case (a).

EXAMPLE:

```
RANK (CARD)
```

where CARD is a string.

A WORD OF CAUTION CONCERNING THE USE OF STRING PROCEDURES

The following remarks apply for all string procedures, whether they appear in programs using the Sort/Merge package or not! (See Chapter XII, PROCEDURES).

If, for example, IN is the name of some arbitrary string procedure, within the procedure body of IN, an operation of the form

```
IN = S
```

where S is a variable whose type is compatible with a string, may be thought of as simply setting a pointer in IN which points to the "string" S. Thus, if the sequence

```
OWN STRING S(6)  $
STRING PROCEDURE IN $
     BEGIN S = 'FIRST' $
          IN = S $
          S = 'LAST'
     END IN $
 WRITE(IN)  $
```

appeared in a program, the "value" written would be "LAST", since IN points to S and since the last "value" of S is "LAST". Also, if S were declared within IN as a non-OWN variable, upon exiting from the procedure S is no longer available.

Thus, here, if IN = S appeared within the procedure body, it is possible that, since IN points to S and since S is not defined outside IN, the result outside the procedure is undefined.

VIII...

ALGOL LIBRARY

Computational procedures for certain standard operations of mathematics and data processing are available for use in Algol programs. A procedure is called by means of the appropriate identifier followed by the arguments that the procedure requires. The call appears in the program in the form:

NAME (<exp_1>, ..., <exp_N>)

where NAME is the identifier of the procedure and <exp_1>, ..., <exp_N> are the desired expressions for the arguments.

The set of standard procedures is divided into two classes: intrinsic functions and library procedures. Included in the latter category is a full complement of Input/Output operations.

INTRINSIC FUNCTIONS

When the programmer calls on an intrinsic function in a source program, the desired operation is made an integral part of the object program by the Algol compiler. Non-intrinsic procedures, rather than being generated in toto by the compiler, are routines that are written separately and linked to the object program only at execution time.

The set of intrinsic functions that are not machine-dependent is discussed and a tabular summary is given below. For a description of intrinsic functions that are dependent on the specific computer and operating system, see Appendix VIII.

ABS (<exp>)

The function ABS has a single argument. The result of the ABS function

is the absolute value of the argument, which may be of any type except BOOLEAN. If the argument is of type INTEGER, REAL or REAL2 the result is of the same type while a COMPLEX argument gives a REAL result and a STRING argument is converted to the corresponding integer value, leading to an integer result.

CLOCK

The function CLOCK has no arguments. The result is of type INTEGER and is the number of seconds elapsed since midnight.

CLOK

The function CLOK has no arguments. The result is of type INTEGER and is the number of sixtieths-of-a-second elapsed since the start of the current year.

DIMENSIONS (<array>)

The result of the DIMENSIONS function is of type INTEGER and its value is the number of dimensions of the parameter <array>.

EVEN (<exp>)

The EVEN function is of type BOOLEAN and the result is TRUE if the parameter is an even number, FALSE otherwise. Before the function is evaluated the parameter <exp> is converted to type INTEGER if it is not already of that type.

LENGTH(<exp>)
LENGTH(<array>,<exp>)

If only one parameter is furnished to the LENGTH function, it must be of type STRING and the result is the number of characters in <exp>.

If the first parameter to LENGTH is an ARRAY then the second parameter must be present to specify which dimension of <array> is being considered. The result is the number of elements in that dimension given by <exp>, i.e., LENGTH(<array>,<exp>) = UPPERBOUND(<array>,<exp>) - LOWERBOUND(<array>,<exp>) + 1 where UPPERBOUND and LOWERBOUND are the library functions described later in this chapter.

The dimensions of an ARRAY are numbered consecutively starting at one. If <exp> is not of type INTEGER it is converted to that type before the function is evaluated. In all cases the result of the function is of type INTEGER.

MOD (<exp$_1$>, <exp$_2$>)

The function MOD requires two arithmetic expressions as arguments. The arguments may be of type INTEGER, REAL, REAL2, or STRING; non-integral expressions will be converted to integer values when the routine is executed. The value of the function is the integer remainder that results from dividing the first argument by the second. The sign of the result is the same as the sign of the first argument.

ODD(<exp>)

The function ODD gives a BOOLEAN result, namely TRUE if the parameter is an odd number, and FALSE otherwise. Before the function is evaluated, <exp> is converted to type INTEGER if it is not already of that type.

SIGN(<exp>)

The function SIGN has a single argument. If the argument is positive the result will be +1; if zero, the result is zero (+ or - according to the sign of the argument); if negative, the result is -1. The type of the argument is INTEGER, REAL, REAL2, or STRING. A STRING argument must represent a string of digits and will be converted to the corresponding integer value.

TABLE OF INTRINSIC FUNCTIONS

Name and Description	Type of argument(s)	Type of result	Examples
ABS(X)=|X|	STRING	INTEGER	
	INTEGER	INTEGER	ABS(-39.2) is 39.2
	REAL	REAL	ABS(<1.0,2.0>) is 2.2360
	REAL2	REAL2	
	COMPLEX	REAL	
CLOCK	none	INTEGER	CLOCK is 3421

TABLE OF INTRINSIC FUNCTIONS (continued)

Namd and Description	Type of argument(s)	Type of result	Examples
CLOK	none	INTEGER	CLOK is 76534
DIMENSIONS(A)	ARRAY	INTEGER	DIMENSIONS(Q) is 2
EVEN(X)	INTEGER REAL REAL2 STRING	BOOLEAN	EVEN(24) is TRUE
LENGTH(A,B)	first: ARRAY second: INTEGER REAL REAL2 STRING	INTEGER	LENGTH(A,B) is 100
LENGTH(X)	STRING	INTEGER	LENGTH(S) is 29
MOD(X_1,X_2) $= X_1-(X_1//X_2)*X_2$	INTEGER	INTEGER	MOD(100,7) is 2 MOD(-200,7) is -4
ODD(X)	INTEGER REAL REAL2 STRING	BOOLEAN	ODD(24) is FALSE
SIGN(X) = 1 X>0 0 X=0 -1 X<0	INTEGER REAL REAL2 STRING	INTEGER INTEGER INTEGER INTEGER	SIGN(34) is 1 SIGN(-1.3) is -1

LIBRARY FUNCTIONS

All library functions can be used in the recursive sense, but for greater object code efficiency the set of library procedures are divided into two classes:

a) Recursive class

b) Normally non-recursive class

The reasoning behind the normally non-recursive class is that most of the standard functions are used in a non-recursive sense. Thus, greater object code efficiency can be obtained by the compiler calling these procedures in a special non-recursive manner.

RECURSIVE LIBRARY PROCEDURES

The set of "recursive only" library procedures is briefly summarized below.

READ (X_1, ..., X_N)	This is the input procedure that is described in Chapters IX and X.
WRITE (X_1, ..., X_N)	This is the output procedure that is described in Chapters IX and X.
MAX (X_1, ..., X_N)	The result of this procedure is the value of the element of maximum value (algebraically) in the list X_1, ..., X_N.
MIN (X_1, ..., X_N)	The result of this procedure is the value of the element of minimum value (algebraically) in the list X_1, ..., X_N.

The input arguments X_I may be expressions, arrays, strings or lists. The functions MAX and MIN are not defined for complex, string or Boolean arguments. The output of MAX and MIN is always REAL.

NON-RECURSIVE LIBRARY PROCEDURES

This set of procedures is further subdivided into two subclasses, namely:

a) Standard mathematical
b) Special

STANDARD MATHEMATICAL PROCEDURES

The standard mathematical procedures require only one input argument and the type of the result depends upon the type of the input argument. This relationship is described below:

Type of Input	Type of Result
INTEGER	REAL
REAL	REAL
REAL2	REAL2
COMPLEX	COMPLEX
STRING	REAL

The mathematical functions that fall into this subclass are:

NAME	DESCRIPTION
ARCCOS (<arith exp>)	inverse cosine
ARCSIN (<arith exp>)	inverse sine
ARCTAN (<arith exp>)	inverse tangent
COS (<arith exp>)	cosine
COSH (<arith exp>)	hyperbolic cosine
EXP (<arith exp>)	exponential to base e
LN (<arith exp>)	natural logorithm
SIN (<arith exp>)	sine
SINH (<arith exp>)	hyperbolic sine
SQRT (<arith exp>)	square root
TAN (<arith exp>)	tangent
TANH (<arith exp>)	hyperbolic tangent

The special procedures that are currently available in the system library are described below. Special procedures that are directly related to the Input/Output routines are described in Chapter IX and X.

NOTE: If an argument to a procedure is required to be of arithmetic type, an argument of type STRING may be used if all its characters are numeric. In this case the string is converted to the associated integer value before the procedure is evaluated.

The following characters are considered to be numeric: plus sign (+), minus sign (-) and blank (△). The blank is also regarded as an alphabetic character.

PROCEDURE CALL	TYPE OF ARGUMENTS	TYPE OF RESULT	DESCRIPTION
ALPHABETIC(X)	STRING	BOOLEAN	Result TRUE if string is alphabetic, FALSE otherwise.
ARG(X)	COMPLEX	REAL	Obtains argument of complex expression X.
CHAIN(X)	INTEGER	none	Used in conjunction with processed MAP (see EXEC II manual); loads and transfers control to program specified by X.
	REAL	none	
	REAL2	none	
	STRING	none	
COMPLEX(X_1,X_2)	either may be		Forms complex number using value of X_1 for real part, value of X_2 for imaginary part.
	INTEGER	COMPLEX	
	REAL	COMPLEX	
	REAL2	COMPLEX	
	STRING	COMPLEX	
COREMAX	none	INTEGER	The result is an upper bound for the number of words of core memory that may be obtained in the next request from the available space pool. The least upper bound depends on the kind of space requested.
CORETOTAL	none	INTEGER	The result is the total amount of core memory remaining in the available space pool. Note that the Algol library allocates space in such a

PROCEDURE CALL	TYPE OF ARGUMENTS	TYPE OF RESULT	DESCRIPTION
			way that at any particular time the value of COREMAX may be considerably less than the value of CORETOTAL.
DATE	none	STRING	The result is a STRING of eighteen characters which gives the month in letters, day of month, and year on which this program was allocated (for absolute programs this is the date when the ABS operation was performed). The result is left-justified and blank-filled.
ENTIER(X)	INTEGER REAL REAL2 STRING	INTEGER INTEGER INTEGER INTEGER	Result is greatest integer in X. Non-integral arguments converted to type INTEGER first.
HEADING (<string>, <arith exp>)	The first is STRING, the second may be INTEGER STRING REAL or REAL2	none	<string> will be printed at the top of each page of output, beginning with the first page that is started after the call. Pages will be numbered consecutively starting with the value of <arith exp>, which will be converted to type INTEGER, if it is not of that type. If the second parameter is omitted, the numbering of pages will not be affected. If both parameters are omitted, the heading action is disabled, i.e., no heading will be printed on succeeding pages.

PROCEDURE CALL	TYPE OF ARGUMENTS	TYPE OF RESULT	DESCRIPTION
IMAG(X)	COMPLEX	REAL	Obtains imaginary part of complex expression X.
INTEGER(X)	INTEGER REAL REAL2 STRING	INTEGER INTEGER INTEGER INTEGER	Rounds X to nearest integer. Result obtained according to the formula ENTIER (X + 0.5)
INTRANDOM(X_1,X_2)	INTEGER REAL REAL2 STRING	INTEGER INTEGER INTEGER INTEGER	Generates next element in pseudo-random number sequence. Result is an integer, the value of which lies between X_1 and X_2 inclusive. An error results unless $X_1 < X_2$.
LOWERBOUND (<array>,<arith exp>)	The first must be an ARRAY, the second may be STRING, INTEGER, REAL, or REAL2	INTEGER	The result is the lower bound for <array> on the dimension specified by <arith exp>.
NUMERIC(X)	STRING	BOOLEAN	Result TRUE if the string X is numeric, FALSE otherwise.
OPTION(<string>)	STRING	BOOLEAN	The function tests for presence of an option letter on the processor call card (including

PROCEDURE CALL	TYPE OF ARGUMENTS	TYPE OF RESULT	DESCRIPTION
			XQT) for the processor currently in effect. The parameter, which should be one character long, specifies the option letter in question. If this letter is present, the result is TRUE, otherwise it is FALSE.
RANDOM(X)	INTEGER	REAL	Generates next element in pseudo-random number sequence. Input argument used only on first call. Sequence starts at INTEGER(X) if X ≠ 0, otherwise at value of CLOK on first call.
	REAL	REAL	
	REAL2	REAL	
	STRING	REAL	
RANK(X)	STRING	INTEGER	See Chapter VII, THE RANK PROCEDURE, for a detailed description
	INTEGER	INTEGER	
	REAL	INTEGER	
	REAL2	INTEGER	
REAL(X)	INTEGER	REAL	Converts X to floating-point form. If X is COMPLEX the result is its real part.
	REAL	REAL	
	REAL2	REAL	
	STRING	REAL	
	COMPLEX	REAL	
REAL2(X)	INTEGER	REAL2	Converts X to double precision floating-point form.
	REAL	REAL2	
	REAL2	REAL2	

PROCEDURE CALL	TYPE OF ARGUMENTS	TYPE OF RESULTS	DESCRIPTION
	COMPLEX	REAL2	
STRING(X)	INTEGER REAL REAL2 STRING	STRING STRING STRING STRING	Converts X to a string of digits. Non-integer expressions are converted to type INTEGER first. If X is negative the result includes a minus sign.
UPPERBOUND (<array>,<arith exp>)	The first must be an ARRAY, the second may be STRING, INTEGER, REAL or REAL2	INTEGER	The result is the upper bound for <array> on the dimension specified by <arith exp>.

THE RANDOM FUNCTION

A function having a result of type REAL and a parameter of type INTEGER is provided in Algol to generate random numbers that are rectangularly distributed on the [0,1] interval. The method used to generate the random numbers is described by the accompanying Algol program. (The FIELD procedure is described in Appendix VIII.)

The parameter supplied is used during the first call only and if non-zero it acts as the starting value of the sequence. Starting values in the neighborhood of 2^{18} give the longest cycles.

```
REAL PROCEDURE RANDOM(I) $
     VALUE I $ INTEGER I $
     BEGIN
          OWN INTEGER LAST $
          OWN BOOLEAN SW $
          REAL R $
          IF NOT SW THEN BEGIN
               SW = TRUE $
               IF I EQL 0 THEN I = CLOK $
               LAST = ABS(I) + (IF MOD(ABS(I),2)EQL 0
                    THEN 1 ELSE 0)
               END $
               LAST = LAST*(3+2**18) $
               LAST = FIELD(LAST,2,35) $
               FIELD(R,1,9) = 128 $
               FIELD(R,10,27) = FIELD(LAST,2,27) $
          RANDOM = 0 + R
          END
```

THE INTRANDOM FUNCTION

The algorithm used by the INTRANDOM is described by the following Algol program.

```
INTEGER PROCEDURE     INTRANDOM(L,U) $
VALUE  L,U $
INTEGER  L,U $
```

```
        BEGIN
        OWN INTEGER LAST $
        OWN BOOLEAN SW $
        IF NOT SW THEN
             BEGIN       SW = TRUE $
                         LAST = 1    END $
        IF U LEQ L THEN ERROR(22) $
        LAST = LAST*(3**22) $

        INTRANDOM = L + ABS(2*(U-L+1)*LAST) $
END          INTRANDOM  $
```

CARDS

PRINTER IX...

PUNCH INPUT

OUTPUT

We now pursue in some detail the subject of communication with the computer. In previous chapters there have been examples of reading in data and printing out answers. In this chapter we present the details (with examples) of the card and printer I/O (Input/Output) processes.

The reader is warned that this chapter is difficult going. In any given explanation, there are usually numerous references to other sections of the chapter. This has been done in order to make descriptions as clear as possible. The reader is advised to read through the whole chapter before trying to 'pin down' any details.

THE READ PROCEDURE

The READ procedure is fully recursive and is used to input information. The general form of the READ is:

READ(<device name>, <label>, <format>, <list name>, <actual input parameter list>

where any combination of the above may be present.

<device name> enables the user to specify any one of the various input devices as the source of input information. The following input devices are available to the user:

CARDS	PCF
CORE	SLIP
DRUM	TAPE
EDIT	

If <device name> is not specified then the source of the input is assumed to be cards. It should be noted that if the <device name> option is used it must be the first name. The other parameters may appear in any order, but the speed of the object program is increased if the above indicated order is observed.

<label> consists of one, two, or three labels (separated by commas if more than one) which are used as alternative exits by the READ procedure.

These exits are utilized when abnormal conditions are encountered while reading. The appropriate exit is chosen in the following manner:

i) Exit is made to the first label (if present) when an EOF (end of file) condition occurs.

ii) Exit is made to the second label (if present) when an EOI (end of information) condition occurs. An EOI condition arises when an attempt is made to read beyond the bounds of the input.

iii) Exit is made to the third label (if present) when an ERR (error) condition occurs. An ERR condition is considered to be either improper input or a malfunction of the input device.

These labels will be referred to as the EOF, EOI and ERR labels, respectively. The standard Algol error exit is supplied for all missing labels. If an exit is made to the Algol error routine, an appropriate message is printed and the program is terminated.

<format> is the name of a sequence of format phrases which has been previously defined in a FORMAT declaration, or is a call on the FORMAT procedure. <format> need not be specified.

<list name> consists of one or more identifiers (separated by commas if more than one) which are the names of lists of variables that have been previously defined by LIST declarations. <list name> need not be specified. Lists and formats are described in greater detail later in this chapter.

<actual input parameter list> is <list elements separated by commas>. See definition of this term in section dealing with the LIST declaration. In general <actual input parameter list> consists of a sequence of identifiers and variables separated by commas. <actual input parameter list> need not be specified.

EXAMPLE

```
REAL A,B,C   $
READ(A,B,C)
```

This reads the value of the three variables A,B,C from card(s). The first number encountered is the value of A, the second is B, and the third is C.

EXAMPLE

```
REAL A,B,C  $
```

```
LIST ZILCH(A,B,C)  $
READ(ZILCH)
```

Exactly the same interpretation as the above example.

THE WRITE PROCEDURE

The general form of the WRITE procedure is

WRITE(<device name>, <label>, <format>, <list name>,
<actual output parameter list>)

where any combination of the parameters may be present.

<device name> enables the user to specify any one of the various output devices for outputting desired information. The following output devices are available to the user:

CORE	PRINTER
DRUM	PUNCH
EDIT	TAPE
PCF	

In the absence of <device name>, PRINTER will be assumed to be the desired device. If <device name> is specified it must be the first parameter. The order of the other parameters to the WRITE procedure is immaterial, but the indicated order will produce greater efficiency in the object program.

<label> consists of one or two labels (separated by commas if two) which are used as alternative exits by the WRITE procedure. These exits are utilized when abnormal conditions are encountered while writing. The appropriate exit is chosen in the following manner:

i) Exit is made to the first label (if present) when an ERR (error) condition occurs. An ERR condition is considered to be a malfunction of the output device.

ii) An EOI condition will cause an exit to the second label if present. If the second label is not present then exit will be made to the first label if it is present. An EOI condition arises when an attempt is made to write beyond the bounds of the output medium.

These labels are referred to as the ERR and EOI labels respectively. The standard Algol error exit is supplied if both labels are missing. If an exit is made to the Algol error routine, an appropriate message is printed and the program is terminated.

<format> is the name of a sequence of format phrases which has been previously defined in a FORMAT declaration, or is a call on the FORMAT procedure. <format> need not be specified.

<list name> consists of one or more identifiers (separated by commas if more than one) which are the names of lists of variables that have been previously defined by a LIST declaration. <list name> need not be specified. Lists and formats are described in greater detail later in this chapter.

<actual output parameter list> is <list elements separated by commas>. See definition of this term in section dealing with the LIST declaration. In gener <actual output parameter list> consists of a list of expressions and/or identifiers. <actual output parameter list> need not be specified.

EXAMPLE

```
REAL A,B,C  $
WRITE(A,B,C)
```

This will write out the values of A,B, and C on the printer.

EXAMPLE

```
REAL A,B,C  $
LIST ROHO(A,B,C)  $
WRITE(ROHO)
```

Same action and interpretation as the above example.

THE FORMAT DECLARATION

We defer the actual definition of a FORMAT declaration until later in this section in the belief that the reader should not be faced with it at this time! The general definition will be much easier to understand if the reader has already seen some particular examples.

Before detailing the uses of the format list let us consider the problem of reading some data cards which should be in a specified format. Suppose that we wish to read in data cards pertaining to student grades. The cards have the following format:

Columns	Contents
1-5	Student Number
6, 7	Initials
8-21	Last Name
22	Status (i.e. class)
23,24	Curriculum
25-37	Not used in this program

Columns	Contents
38-44	Course designation
45, 46	Not used in this program
47	Credit hours
48-59	Not used in this program
60	Letter grade

Assume that the columns not used have information in them which is of no concern to the problem at hand. The problem is now to read in the above data in a form which will make the manipulations easy and still enable us to print out all of the above data. It is this type of problem which gives rise to the necessity of specifying the card format.

For this problem it is reasonable that we would like the computer to take the following action with regard to card reading:

1. Read the card (i.e. Activate the input card reader)
2. Accept the first five columns as an integer (student number)
3. Accept the next two columns as a string (initials)
4. Accept the next fourteen columns as a string (last name)
5. Accept the next column as an integer (status)
6. Next two columns as integer (curriculum)
7. SKIP the next thirteen columns (not used)
8. Next seven columns as a string (Course title)
9. SKIP two columns (not used)
10. One column as an integer (credit hours)
11. SKIP twelve columns (not used)
12. One column as a string (letter grade)

The format declaration is used to take care of all of the above functions. For this example the format would be:

```
FORMAT UNAMEIT (A,I5,S2,S14,I1,I2,X13,S7,X2,I1,X12,S1)  $
```

Note that there is one entry in the format for each numbered line above. Each of the items in the above format is referred to as a "format phrase". A reasonable program segment for the above problem would be:

```
INTEGER STUDENTNUMBER, STATUS, CURRICULUM, CREDIT$
STRING INITIALS(2), LASTNAME(14), COURSE(7), GRADE(1)$
FORMAT UNAMEIT (A,I5,S2,S14,I1,I2,X13,S7,X2,I1,X12,S1) $
LIST EVERGREEN(STUDENTNUMBER,INITIALS,LASTNAME,STATUS,CURRICULUM,
     COURSE,CREDIT,GRADE)$
READ(UNAMEIT,EVERGREEN)
```

The list specifies what is to be read, while the format specifies how the reading is to take place. Formats for outputting information are constructed in a similar fashion.

Now that we have a feeling for how formats are used, we proceed to define exactly what is meant by a FORMAT declaration.

The general form of a FORMAT declaration is:

FORMAT <format name> (<format phrases separated by commas>)$

where <format name> is a name supplied by the programmer and <format phrases separated by commas> refers to the uses of the format phrases described below. Note that the descriptions given below are only for the PUNCH, PRINTER and CARDS devices. Formats may be used with some of the other devices and the interpretation of the format by these devices is described in the relevant sections of the manual.

FORMAT PHRASES

Ab.a CARDS Activate the card reader. The b.a is ignored. The activation must precede the format list which applies to the card being read.

PRINTER Activate the output device skipping b lines before printing and a lines after printing. The activation must follow the format list which applies to the line being printed. Note that upon completion of printing a line, the printer does NOT advance the paper unless it is told to by a non-zero a value. The .a may be omitted if one desires a to be zero, i.e., A3 is the same as A3.0.

PUNCH Activate the punch. The b.a part is ignored. Also the punch only punches 80 columns and any output beyond 80 columns is lost. The activation must follow the format list which applies to the line being punched.

Bw CARDS Accept a BOOLEAN variable from w consecutive columns. This phrase translates a string (without quotation marks) into a Boolean expression. Only the first character of the string is interpreted and should be T or F.

PRINTER

PUNCH Output a Boolean expression using w columns. The words TRUE or FALSE will be outputted if space permits. Otherwise the w leftmost characters of the word will be outputted.

Cw.d CARDS Read a COMPLEX number from 2w+3 columns. In each w field,

the number is treated identically as if the format phrase was Rw.d. The two fields are separated by a slash (/) and bounded by corner brackets (< >). For example 3+2i would be punched on card as

<3.0/2.0>

and the proper format phrase would be

C3.1

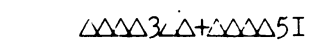
where the complex number occupies 2*3+3=9 columns on the card. If this format is objectionable, then the user can format each part of the complex number separately using any other phrases. For example, the format

F(A,I5,X3,I5)

and the READ procedure call (where C is a complex number)

READ(F,C)

and the data card



will set C to the value <3.0, 5.0>

PRINTER

PUNCH Output is a COMPLEX number using 2w+3 columns. This phrase is equivalent to the set of format phrases.

'<', Rw.d, '/', Rw.d, '>'

If this format is objectionable, then the user can format each part of the complex number separately using any of the other phrases. For example, the format

F(I5,X2,'+',I5,'I',A1.1)

and the WRITE procedure call

WRITE (F,<3.0,5.0>)

will produce the line image

3 + 5I

Dw.d CARDS Same action as Rw.d

PRINTER

PUNCH Using w columns, output a number (REAL) with d digits to the right of the decimal point, locating the decimal point properly, e.g., if a variable has the value 3.145×10^2, a D5.1 format will output

314.5

En CARDS Not allowed.

PUNCH Ignored.

PRINTER Eject the page to line number n-1 and start the next activation

phrase at this point. A standard page (66 lines) consists of top and bottom margins of six lines each and a body of 54 lines. A standard page can be modified using the MARGIN procedure (q.v.). In any case, if the body currently contains K lines, then these lines are numbered from 0 to K-1. This phrase will then cause the page to be ejected to line n-1 mod K if $n \neq 0$ or line K-1 if $n = 0$. No line will be printed. Except for ejecting the page, this phrase produces no effect on the formatting process.

Fw CARDS Read the next w columns as if there were no format (i.e. in free format). The format phrase, F80, will take longer to read a card than no format at all. The F format cannot extend from one card to another. Free format is explained in the section describing the device CARDS.

PUNCH

PRINTER Not allowed.

Iw
Iw.d CARDS Read an integer expressed to the base d from w consecutive columns. The value of d may be

$$d = 0 \quad \text{or} \quad 2 \leq d \leq 10.$$

If d is omitted or d = 0 then the base is assumed to be 10. Blanks occurring between the end of the number and the end of the field are interpreted as zeros.

PRINTER

PUNCH Output an integer expressed to the base d using w columns. The value of d may be

$$d = 0 \quad \text{or} \quad 2 \leq d \leq 10.$$

If d is omitted or d = 0 then base ten is assumed. The number is right justified in the field.

Ow
Ow.d CARDS Not allowed.

PRINTER

PUNCH The same as Iw.d except that the sign bit (leftmost bit) is treated as a numeric bit, i.e., the word is considered to be a 36-bit binary number. If the number is not large enough to fill the field then zeros are inserted to the left of the left-most digit.

Pw CARDS Sets the scanner to start scanning at column w of the present card image beginning with the next format phrase. The range of

w should be $1 \leq w \leq 80$.

PRINTER

PUNCH Sets the output editor to start creating a line image at column w of the current line image. The range of w should be

$$1 \leq w \leq 132$$

Rw.d CARDS Accept a REAL variable from w columns - If one of the columns contains a decimal point, then its position is used to define the actual decimal point - If there is no decimal point explicitly punched in the card then it is assumed to be located d digits to the left of the right-hand end of the field giving the numerical value. For example, R5.2 will force the following correspondences:

CARD	COMPUTER
12345	123.45
12.34	12.34
12,03	0.12×10^3

Note that the character sequence "03" serves to multiply the value by ten with a power of 03.

PRINTER

PUNCH Output a REAL variable using w columns with d-1 digits after the decimal point. The form of the output is always:

x.xx, ee (represents R9.3)

where the first x is not zero (unless all of the x's are zero) and ee represents the exponent of ten. Since one must leave room for the sign of the mantissa, sign of the exponent and decimal point the safe rule is that w must be equal to or greater than d+7. Note that d may not be zero.

Sw CARDS Read in a STRING from w consecutive columns. The string is left justified in the input string variable and any remaining positions in the string variable are "space" filled. A string may spill over from one card to the next and still be processed properly. No quote marks are needed to delimit the string.

PRINTER

PUNCH Output a STRING using w columns.

Tw.d CARDS Same action as Rw.d.

PUNCH

PRINTER Output a REAL variable using w columns and truncate the number to d significant figures while inserting the decimal point in its proper place. e.g. T6.3 will do:

COMPUTER	PRINTED PAGE
123.456	123.
0.002345	.00234
-0.002345	****** (i.e. field is too small)

Uw

Uw.d CARDS Not allowed

PUNCH

PRINTER Same as Iw.d except that the sign bit (leftmost bit) is treated as a numeric bit, i.e., the word is considered as a 36-bit positive binary number rather than a 35-bit binary number with sign.

Xw CARDS Skip over (X-out) w columns.

PUNCH

PRINTER Skip over w columns. Unless the Pw format phrase is used this is equivalent to inserting w blanks.

Zw

Zw.d CARDS Not allowed.

PUNCH

PRINTER The same as Iw.d except that the field is filled at the left with leading zeros, if necessary. If the integer is positive, w digits are outputted, otherwise w-1 digits preceded by a minus sign are outputted.

'(quotation mark)

CARDS Not allowed.

PUNCH

PRINTER Quotation marks must appear in matched pairs, i.e., there must be a "left" and a corresponding "right"quote mark in each use of the quotation marks. All characters, including spaces, between the quotation marks will be outputted exactly as written. The only way to output a quotation mark is to use the pound sign ahead of it.

Rules Concerning Format Phrases

1. If a format phrase can be of the form mw.d where m is any letter, then w and d must be ≤ 63. If a format phrase can only be written in the form mw then w must be ≤ 4095. For example, for the format phrase Iw, w must be ≤ 63 since Iw.d is a permissible form but for Sw, w may be as large as 4095.
2. w and d are always considered to be unsigned numbers.
3. When a COMPLEX variable or expression, <exp>, occurs as a parameter to READ or WRITE and the associated format phrase is not Cw.d, the <exp> is considered to be the two REAL variables, REAL (<exp>) and IMAG(<exp>). Thus on input two values are read and converted to COMPLEX and on output a COMPLEX number is decomposed into two REAL values.

REPEATED FORMAT EXPRESSIONS

It is frequently desirable to repeat one (or several) pieces of format without the programmer writing the whole thing out in laborious detail. These situations are taken care of by three cases:

1. Definite repeat
2. Variable repeat
3. Indefinite repeat

DEFINITE REPEAT

For example if it is desired to print out ten REAL variables on one line where each variable is to be formatted by R12.4, one could write:

```
FORMAT  OWW(R12.4,R12.4,R12.4,R12.4,R12.4,R12.4,R12.4,R12.4,
          R12.4,R12.4,A1) $
```

However this is very inconvenient. But by using the definite repeat, the equivalent format

```
FORMAT  OWW(10R12.4,A1) $
```

can be written. Now consider the case where we wish to repeat two variables per line on the page for 5 lines of print. We merely write

```
FORMAT  GRUNST (5(2R12.4,A1)) $
```

Thus the definite repeat really assumes two forms:

First form: <integer giving number of repeats> <format to be repeated>

Second form: <integer giving number of repeats>(<format phrases separated by commas>)

VARIABLE REPEAT

Let's assume that we have a program which is to print out several lines and we do not wish to specify until run time how many numbers to print on each line. (We could repunch the format declaration prior to running the program each time but this is undesirable.) In other words we have available and wish to use some expression or variable which specifies how many variables we desire on the line. The variable repeat is used in this instance to allow an expression to specify how many times a (sequence of) format phrase(s) is to be repeated.

Before the exact form of the variable repeat is given, let us illustrate its use.

```
INTEGER   I,N$
REAL ARRAY X(1:1000)$
FORMAT  GUDGEON(A,X5,*R8.4)$
READ(N)$
READ(GUDGEON,2*N+1, FOR I = 1 STEP 1 UNTIL 2*N+1 DO X(I) )
```

The first call on READ will cause a value for N to be assigned from the first card of input data. The second call on READ will cause:

1) The value of 2*N+1 to be substituted for the *. If N were 4 then the format becomes A,X5,9R8.4.
2) A card will be read.
3) The first five columns of the card are skipped.
4) The number in columns 6-13 inclusive will be placed in X(1), the number in columns 14-21 inclusive will be placed in X(2), etc., intil 2*N+1 values have been read.

Note that in the second READ statement, a value is not read from cards for 2*N+1 since it corresponds to the variable repeat (*) and is only used to determine the repeat value. Also the variable repeat only affects the number of times the format is repeated.

The form of the variable repeat is then:

*<single format phrase> OR *(<format phrases separated by commas

Suppose n asterisks occur in a FORMAT declaration. Then in the READ or WRITE procedure in which this format is a parameter, the first n expressions in either the input or output list are considered as call-by-value expressions and the value of each is associated from left to right with the respective variable repeat. The values of the variable repeats are then retained

throughout the remainder of READ or WRITE. These first n expressions are used only to give values to the variable repeats and are ignored during the remainder of the processing of the procedure. These first n expressions must be of INTEGER type and have non-negative values. If the value of any variable repeat is zero then the associated (sequence of) format phrase(s) is skipped. It should also be noted that if these n expressions are used in a READ statement, then the expressions are evaluated before processing the input variables.

INDEFINITE REPEAT

The form of the indefinite repeat is:

(<format phrases separated by commas>)

The indefinite repeat is used when the user does not wish to specify how many times a (sequence of) format phrase(s) is to be repeated. The indefinite repeat will cause <format phrases separated by commas> to be repeated until the input or output list is exhausted. Thus the entire format is viewed as an indefinite repeat. No format phrases to the right of the first indefinite repeat will ever be interpreted. This rule has the following implications:

1. An activation phrase should always be included in an indefinite repeat.
2. READ terminates immediately upon exhausting the input list regardless of which format phrase is being interpreted.
3. WRITE terminates if the output is exhausted and all of format phrases between the point in the format at which the output list became exhausted and the end of the indefinite repeat have been interpreted. For phrases requiring output for which no output exists, blanks are inserted instead.

EXAMPLE:

```
INTEGER M,N,P$
READ(M,N,P)$
BEGIN
INTEGER ARRAY D(1..20,1..10)$
REAL ARRAY A(1..M), B(1..N), C(1..P)$
FORMAT PUMPERNICKEL(A,*R12.4,A,*R10.3,A,*D5.1,(A,10I4))$
LIST  BORSCHT(M,N,P,A,B,C,D)$
READ(PUMPERNICKEL,BORSCHT)
END
```

In the preceding example, the values of M,N,P are assigned to the asterisks in the FORMAT from left to right, M elements of A are read under R12.4, N elements of B are read under R10.3, P elements of C are read under D5.1, and the array D is completely filled via the indefinite repeat (A,10I4)

EXAMPLE:

```
REAL Y$
FORMAT MOXIE(' X    SQRT(X)', A1, (D3.1,X2,T8.6,A1))$
LIST AXOLOTL (FOR Y=1 STEP 1 UNTIL 6 DO (Y,SQRT(Y)))$
WRITE(MOXIE,AXOLOTL)
```

The above program fragment will result in:

```
  X       SQRT(X)
1.0       1.00000
2.0       1.41421
3.0       1.73205
4.0       2.00000
5.0       2.23606
6.0       2.44948
```

Note: Any of the various repeat options are considered to be single phrases.

THE FORMAT PROCEDURE

The use of a <format name> in a READ or WRITE procedure may be replaced by a call on the FORMAT procedure of the form:

FORMAT (<string expression>)

This call results in the string expression being processed at this point as a format string. The value of the string expression must have the form:

(<format phrases separated by commas>)

to be interpreted properly as a format string. Note that the outermost parentheses must be present in the string. It should also be noted that this string expression can be read in from cards as well as generated by any of the readily available string operations.

GENERAL REMARKS ON FORMATS USED WITH WRITE

1. A format describes how the user wants the outputted line to appear, where the descriptive process generally proceeds sequentially from

left to right (with the exception of the Pw phrase). Each device has a maximum line length which it will accept (the longest being 132). Any characters specified in excess of the maximum number will be lost and no error message will be produced. Any columns not explicitly filled by the user will be filled with blanks.

EXAMPLE

```
FORMAT KNIF(X120, 'SOME OF THIS ISNT', A1) $
WRITE(KNIF)
```

The output on the high speed printer page (128 character line length) for the above fragment of program will consist of 120 blanks followed by

SOME OF

2. A plus sign is never outputted by any of I,T,D,R,C,Z,O,U.
3. BOOLEAN variables used for output have only two possible values:

   TRUE

   and

   FALSE

   thus any use of the B format must use at least 4 and probably 5 as the number of columns, if the entire word is to be printed.
4. If the field is too small for any of the format phrases, then one of the following actions will occur.
   a) If the format phrase is Sw or Bw, the output will consist of as many characters as possible, starting with the leftmost character.
   b) If the format phrase is Iw.d, Ow.d, Uw.d or Zw.d then the most significant part is lost, the sign (if present) is retained and the resulting field is preceded by an asterisk (*).
   c) If the format phrase is Rw.d, Iw.d or Dw.d, then the number will be truncated to fit the field. If the field is too small for even this minimum information, an error message will be produced consisting of w asterisks (*) showing the requested size of the field.
   d) If the format is complex, no check is made on the entire field size but only on each sub-field and the errors will be indicated as in c) above.
   e) If none of the above conditions are true then an appropriate error message is printed and the program is terminated.
5. All output (except STRING and BOOLEAN values) is always pushed over to right (right justified) within the available column specification.

EXAMPLE:

		PRINTER OR PUNCH
63.4	under R9.2	6.3,+01
-63.4	under R9.2	-6.3,+01
0.00634	under R9.2	6.3,-03
-0.00634	under R9.2	-6.3,-03
-127	under I9	-127
TRUE	under B9	TRUE

6. In the Iw.d and Uw.d formats all leading zeros of the field are deleted.
7. The following program fragments are equivalent.

```
ARRAY  A(L1..U1,L2..U2,...,Ln..Un) $
WRITE (FOR I1 = L1 STEP 1 UNTIL U1 DO
          FOR I2 = L2 STEP 1 UNTIL U2 DO
                 :
          FOR In = Ln STEP 1 UNTIL Un DO
              A(I1,I2,...,In))
```

and

```
ARRAY A(L1..U1,L2..U2,...,Ln..Un) $
WRITE (A)
```

For two dimensional arrays, this amounts to outputting the array row by row. The second fragment executes far faster than the first.

8. Allowable conversions of type in WRITE with format are given in the following table:

FORMAT	ALLOWABLE TYPES OF PARAMETERS
B	BOOLEAN
C	COMPLEX
D,R,T	INTEGER REAL BOOLEAN REAL2 COMPLEX (only the real or imaginary part will be used. For complete explanation see discussion of Cw.d format phase.)
I,O,U,Z	INTEGER REAL BOOLEAN STRING REAL2 COMPLEX (same as D format)
S	STRING

GENERAL REMARKS ON FORMATS USED WITH READ

1. An activation phrase (A) must be the first format phrase in a format that is a parameter to READ. If the first phrase is not an activation then an error message will be given and the program terminated.
2. A complete discussion of free format is given in the section describing the procedure CARDS. The free format specified by the Fw phrase differs from free format in the following ways:
   a) With the format Fw, an activation phrase must be specified.
   b) With the format Fw, an asterisk (*) as data is considered an illegal character unless it is part of a string constant.
3. By using a format in the READ procedure, a card can be considered to have up to 4095 columns. This feature is invoked when any format phrase (except X, P and F) extends beyond the end of a card. When this occurs,

column one of the next card immediately follows column 80. No activation phrase is necessary. The phrases X, P and F may never run across card boundaries.

EXAMPLE

```
STRING   T(180) $
FORMAT   F(A,S180) $
READ     (F,T)
```

After reading is initiated by the activation (A) phrase, the S format will read two additional cards in order to reach the end of the specified field. The string T is filled with the first two cards and the first twenty columns of the third.

4. If any part of a card has not been examined when the input parameter list is exhausted, any information on that part of the card is lost. In the above example, columns 21 through 80 of the third card are discarded.

5. The following two program fragments are equivalent.

```
ARRAY  A(L1..U1,L2..U2,...,Ln..Un) $
READ   (FOR  I1 = L1 STEP 1 UNTIL U1 DO
            FOR I2 = L2 STEP 1 UNTIL U2 DO
                    .
                    .
                FOR In = Ln STEP 1 UNTIL Un DO
                    A(I1,I2,...,In) )
```

and

```
ARRAY  A(L1..U1,L2..U2,...,Ln..Un) $
READ   (A)
```

For two dimensional arrays this amounts to filling the array row by row. The second fragment will execute far faster than the first.

6. Conversion of type in a READ procedure with format is a very intricate process involving the interaction of the format, the type of parameter and the type of data on the card. This process is described here.

A datum on a card will be called a number except for strings. A number is BOOLEAN if it begins with a T or an F. A number is an integer if

i) there is no decimal point in the number and

ii) there is no exponent present in any form and

iii) if the format is Cw.d, Dw.d, Rw.d or Iw.d then d = 0.

Any other occurrence of a number is considered a real number. All real numbers are double precision.

Any conversions necessary are made using the usual transfer functions. The allowable type conversions are

a) If the number is Boolean then the parameter must be BOOLEAN.

b)

FORMAT	ALLOWABLE TYPES OF PARAMETERS	
C	INTEGER, REAL, REAL2	Only the real part is used. The imaginary part is discarded
	COMPLEX	
	BOOLEAN	Only the real part is examined. If it is an integer number and has value of 0 or 1 then it is an allowable conversion
D,R,T	INTEGER	
	REAL	
	REAL2	
	BOOLEAN	Allowed only if number is integer and has value 0 or 1.
	COMPLEX	Only half of a complex number will be assigned a value. See discussion of Cw.d format phrase and remark 3 on formats.
B,I	INTEGER	
	REAL	
	REAL2	
	BOOLEAN	Allowed only if number is integer and has value 0 or 1.
	COMPLEX	Only half of a complex number will be assigned a value. See discussion of Cw.d format phrase and remark 3 on formats.
S	STRING	

THE LIST DECLARATION

The general form of the LIST declaration is:

LIST <list name>(<list elements separated by commas>)$ where <list name> is a name supplied by the programmer and <list elements separated by commas> is discussed below.

First we point out that any LIST which can be used in READ may also be used in WRITE -- but it is not true that any LIST usable in WRITE may be also used in READ.

A list element may assume any of the following forms:

1.	A simple variable name	(e.g. X)
2.	A subscripted variable	(e.g. A(I,J))
3.	An ARRAY name	(e.g. A)
4.	A STRING element	(e.g. NAME(2))
5.	A STRING name	(e.g. NAME)
6.	A sub-STRING	(e.g. NAME(6,9))
7.	A LIST name	(explained below)
8.	A generative process	(explained below)
9.	An intrinsic or predefined procedure call	(e.g. SQRT(X))
10.	An explicitly defined procedure call	(e.g. MYOWN(A,B,P,Q))
11.	An arithmetic expression	(e.g. X+Y)
12.	A Boolean expression	(e.g. A IMPL B OR C EQUIV P)
13.	An arithmetic constant	(e.g. 5.98734)
14.	A STRING constant	(e.g. 'FRITZWEG')
15.	A BOOLEAN constant	(e.g. TRUE)

Any of the first six forms of a list element (and with certain reservations, the seventh and eighth may be used in a LIST to be used in READ. Any of the forms may be used in WRITE.

Thus we now see that back at the beginning of this section where we talked about <actual input parameter list> we meant that allowable items in the list are any of the first six (and with reservations, the seventh and eighth) of the list element forms.

Also where we mentioned <actual output parameter list> we now define the elements to be of any of the list element forms enumerated above.

A generative process (item 8 in the enumeration above) is of the form:

FOR <variable identifier or subscripted variable> = <for list> DO (<any of the fifteen forms of list element separated by commas>) where <for list> is defined as:

<arithmetic expression>

or

<arith exp> STEP <arith exp> UNTIL <arith exp>

or

<arith exp> WHILE <Boolean expression>

or

any sequence of the above separated by commas. See the section describing the FOR statement for a complete explanation.

EXAMPLE:

```
INTEGER I,N $
READ (N) $
BEGIN
ARRAY A,X(1..N) $
LIST ELDIABLOROJO(FOR I = 1 STEP 1 UNTIL N DO (A(I), X(I)))$
READ(ELDIABLOROJO) $
WRITE(ELDIABLOROJO)
END
```

The above program fragment reads in a value for N, establishes the size of arrays A and X, reads in values for

$$A_1, X_1, A_2, X_2, \ldots, A_N, X_N$$

and will then write out the values in the same sequence.

By now it is no doubt clear to the reader that the form of the generative process which is acceptable to READ is

FOR <variable identifier or subscripted variable> = <for list> DO (<any of the first six forms (and with reservations, the seventh and eighth) of the list element forms separated by commas>).

A LIST may itself be used as a list element in a LIST (i.e. LIST's may be nested). If the parent LIST is used in a READ procedure then all list elements of all nested LIST's should follow the above suggestions regarding the allowable forms of list elements.

Multiple LIST's may be used in input/output statements. Therefore the parameter <list name> in a READ or WRITE procedure may in fact consist of one or more LIST's.

EXAMPLE:

```
LIST L(H,J) $
LIST X(H,J,L) $
LIST Z(A,B,C,X) $
A = 1 $ B = 2 $ C = 3 $ H = 7 $ J = 8 $
WRITE (L,L,L,L) $
WRITE (Z)
```

The first WRITE procedure contains multiple LIST's; and the second uses a nesting of LIST's that is three deep.

THE CARDS DEVICE

CARDS may only be used as a parameter to READ and is used to specify the on-line card reader as the <device name>. This is exactly equivalent to omitting <device name>. The cards to be read should follow the control card which caused execution of the program and should not include control cards other than EOF cards.

The <format> parameter to the READ procedure is optional for CARDS and should be used only when the user desires to make a rigid specification of the card contents. When a format is specified, the operation is as described previously. Otherwise the cards are read in a so-called "free format" (described below).

If alternative exits are provided to the READ procedure (as the parameter <label>), then for the <device name> CARDS these exits are taken in the following circumstances.

1) Exit is made to the EOF label if an EOF control card is encountered and the input parameter list has not been exhausted. Subsequent calls on READ will continue to read the cards following the EOF card.
2) Exit is made to the EOI label if a control card other than an EOF card is encountered and the input parameter list has not been exhausted. Any subsequent calls on READ will continue to exit to the EOI label.
3) Exit is made to the ERR label only if a card contains a sequence of characters which cannot be transformed into an acceptable value for the associated input variable.

This might be due to a mispunched card, a misread card, or a request for an illegal type conversion.

FREE FORMAT WITH CARDS

Data cards to be read in free format whould be punched according to the following rules. Four types of data may appear:

1) An integer value is represented by a sequence of digits, optionally preceded by a sign.
2) A real value may be represented in any of several forms:
   a) A sequence of decimal digits containing a decimal point
   b) An ampersand (&) followed by an unsigned or signed integer. This represents a power of ten
   c) A sequence of decimal digits with or without a decimal point followed by a power of ten. This power of ten may have the form b above or the form comma (,) followed optionally by a blank, or sign, followed by an unsigned integer
   d) Any of a, b, or c preceded by a sign.
3) A string value is represented by a sequence of characters not containing an apostrophe (') and enclosed in apostrophes used as quote marks. IMPORTANT: The characters # and : do not have the special significance which is attached to them within string constants in an Algol program. As a result, it is not possible to read the character apostrophe (') under free format.
4) A Boolean value is represented by a sequence of characters except blank and asterisk, the first of which is either T or F. Only the first character is examined to determine the value: TRUE for T, FALSE for F.

EXAMPLES:

Data on Card	Type	Value
928	INTEGER	+928
-2	"	-2
+6.5	REAL or REAL2	+6.5
3.2&2	REAL or REAL2	$+3.2 \times 10^{+2}$
&-3	" " "	$+1.0 \times 10^{-3}$

Data on Card	Type	Value
5.4,3	REAL or REAL2	$+5.4 \times 10^{3}$
5.4, 3	" " "	$+5.4 \times 10^{3}$
5.4,+3	" " "	$+5.4 \times 10^{3}$
5.4,-3	" " "	$+5.4 \times 10^{-3}$
'TOM'	STRING	TOM
TOM	BOOLEAN	TRUE

If an asterisk (*) not contained in a string constant occurs on a data card then the rest of the card is ignored.

If several values appear on the same card, they must be separated by one or more blanks. A blank may not appear in a data item except in a string value and in a real value following a comma. In the latter case, only a single blank is permitted, and it is exactly equivalent to a plus sign. A value is always terminated after column 80, and therefore a value may not extend from one card to the next. An asterisk not contained in a string value will act the same as the end of card.

At least one card is always read (unless an EOI condition exists). Additional cards will be read until enough values are obtained to exhaust the input list. Unused values on the last card are lost since the next call on READ begins by reading a card.

Each INTEGER, REAL, REAL2, BOOLEAN, STRING simple variable and element of an array requires one value from the input cards. Each COMPLEX variable and element of an array requires two REAL values, which become respectively the real and imaginary components of the complex number. The corner bracket form of complex numbers may not be used for free format input. Values representing REAL values are always read as REAL2 values and then truncated to REAL, if necessary

The following table indicates the permitted types of input parameters for each type of value which may appear on a card.

Type of Value on Card	Allowable Types of Input Parameters	
INTEGER	INTEGER	
	REAL	
	REAL2	
	COMPLEX	Two values are read
	BOOLEAN	Convert 0 to FALSE, 1 to TRUE

Type of Value on Card	Allowable Types of Input Parameters	
REAL or REAL2	INTEGER	
	REAL	
	REAL2	
	COMPLEX	Two values are read
BOOLEAN	BOOLEAN	
STRING	STRING	

THE PRINTER DEVICE

The WRITE procedure outputs to this device when the <device name> parameter is PRINTER or is omitted. Output is printed on the same printer as the rest of the user's listing.

None of the alternative exits specified by <label> are used when output is to the printer.

The <format> parameter is optional. When a format is supplied, operation is as previously described. If a format is not supplied by the user, the implied format described below is used.

Type of Output Value	Implied Format Phrase
INTEGER	I12
REAL	R12.5
REAL2	R12.5
COMPLEX	Two REAL values are printed, each under R12.5
BOOLEAN	B12
STRING	See below

Each call on WRITE begins a new line. Except for type STRING, ten values are printed per line until the output parameters from this call on WRITE have all been printed. If less than ten values are printed on the last line then blanks are supplied to complete the line. A STRING value, however, always begins a new line, after forcing any values preceding it in the output list to be printed. The string is divided into substrings of 132 characters each, with blanks supplied to complete the last substring if necessary, and these substrings are printed, one per line. Any output parameters following the string will begin a new line.

The user should note that two different models of printer are in use. A 1004 printer accepts a line of 132 characters. The high speed printer accepts a line of 128 characters, and thus some characters of long strings

printed under implied format will be lost.

AUXILIARY PROCEDURES TO CONTROL PRINTER

Several procedures are available for providing additional control over the printer and are described below.

The HEADING Procedure

This procedure may be called with 0, 1, or 2 parameters:

HEADING

HEADING (<exp$_1$>)

HEADING (<exp$_1$>,<exp$_2$>)

where <exp$_1$> should be of type string and <exp$_2$> should be of type integer and should be in the range $0 \leq$ <exp$_2$> ≤ 4095. The top margin of each succeeding page will contain a heading line composed of the first 60 characters of <exp$_1$>, the current date and the page number. If <exp$_2$> is present and is non-zero,then the next page heading printed will contain this page number. Otherwise the page number is not altered. The form without parameters will terminate the printing of the heading, including page numbers. Note that if the top margin is not large enough to contain the heading (because of a call on MARGIN) then no heading will be produced.

The MARGIN Procedure

The form of a call on this procedure

MARGIN (<exp$_1$>,<exp$_2$>,<exp$_3$>)

where each parameter may be of type INTEGER, REAL, REAL2, or STRING. Conversion of the value of each parameter to type INTEGER is made in the usual manner. Thereafter, each page is considered to have a top margin of <exp$_1$> lines, a body of <exp$_2$> lines, and a bottom margin of <exp$_3$> lines. Note that, in general, the position of the paper in the printer following a call on margin will be undefined. However, if the standard margin of 6, 54, 6 is called for, the position will be adjusted so that the perforations of standard paper will separate pages.

The PAPER Procedure

This procedure is called by:

PAPER (<exp>)

where <exp> may be of type INTEGER, REAL, REAL2, or STRING. The value of

<exp> will be converted to type STRING in the usual manner. During printing of the output, <exp> will be printed at the top of the next page, a page will be ejected, and the printer will be suspended. This feature is used when special forms are to be inserted in the printer. An unsolicited key in (ΔPR4 for the 1004, ΔPR1 for the high speed printer) is necessary to continue printing (see Exec III manual).

THE PUNCH DEVICE

Output may be made to the on-line card punch if the <device name> parameter of WRITE is PUNCH. The card punch accepts a line of 80 characters, and any excess characters will be ignored.

None of the alternative exits specified by <label> are used when the output device is the card punch.

The user may specify a particular format for the output and operation will be as previously described. If the <format> parameter of WRITE is omitted, the implied format described below is used for each type of output value.

Type of Output Value	Implied Format Phrase
INTEGER	I16
REAL	R16.8
REAL2	R16.8
COMPLEX	Two REAL values are punched, each under R16.8
BOOLEAN	B16
STRING	See below

Except for type string, five values are punched per card until all values have been outputted. If less than five values are are available for the last card for this WRITE, blanks are supplied. A STRING value always begins a new card, after forcing any preceding output parameters to be punched. The string is divided into substrings of 132 characters, and the first 80 characters of each substring are punched, without quote marks, into a card. Any output values following the string will begin on a new card. Note that it is usually undesirable to punch strings longer than 80 characters under the implied format.

LIBRARY PROCEDURES FOR CARD, PRINTER AND PUNCH I/O

Procedure call	Type of argument(s)	Type of result	Description
CARDS	none	none	Defines card reader as input device in a READ procedure call.
FORMAT (<exp>)	STRING	FORMAT	Converts <exp> into an edited list of format phrases.
HEADING (<exp_1>, <exp_2>)	First is STRING, second is INTEGER	none	Prints <exp_1> at the top of every page of printed output. <exp_2> controls page numbering.
MARGIN(<exp_1>, <exp_2>,<exp_3>)	Any may be INTEGER, REAL, REAL2, STRING	none	Defines form of printed page: <exp_1> is number of lines in top margin; <exp_2> is number of lines in body of page; <exp_3> is number of lines in bottom margin. Non-integer arguments are converted. The standard values are 6,54,6 respectively.
PAPER(<exp>)	STRING	none	<exp> is printed at top of next page, the page is ejected and the printer is then suspended. This facilitates the changing of paper forms.
PRINTER	none	none	Defines printer as output device in a WRITE procedure call.
PUNCH	none	none	Defines punch as output device in a WRITE procedure call.

TAPE

DRUM X...

INPUT

OUTPUT

This chapter discusses the I/O devices TAPE and DRUM and the procedures that relate to the use of these devices. To understand this chapter the reader should know exactly what magnetic tape and drum are and the basic dynamics of their use. Also the reader should be familiar with the previous chapter dealing with card and printer I/O.

THE TAPE DEVICE

TAPE is a non-recursive procedure that specifies a <device name> to the I/O procedures READ, WRITE, POSITION and REWIND. Such a specification selects one of the magnetic tape units as the input/output device. To use TAPE as a parameter to any of these procedures (which is the only allowed use of TAPE) it must have the form

TAPE(<exp>)

where <exp> must be an expression that can be converted to type STRING and have the value

'<unit>' or '<unit> <numeric>'

<unit> denotes the desired logical unit and <numeric>, if present, denotes the length of the tape in feet. If <numeric> is omitted, then, for magnetic tape, the length is taken to be 1200 feet. If <numeric> consists of all blanks then the length is set to zero. The length of the tape is set every time <numeric> is specified regardless of the position of the tape. The assumed length is only made at the time of the first reference to a particular tape unit. The tape length is used to allow the user to make a programmed recovery when the physical end of tape is encountered while writing. This is

done because some tape units (such as Univac II A) will not, by themselves, alert the programmer to the end of tape condition. The procedure TAPE will keep track of how much tape has been written and, if the tape length was correctly specified, give an EOI condition when the physical end of tape is near.

The following table indicates allowable values of <unit> and the device <unit> specifies for each value.

Value of <unit>	Physical Device Specified
A - Z	Tape units A - Z
0	Printer
1	Card reader
2	Punch
4 - 9	Drum simulated tapes

Drum simulated tapes are discussed in the next section. If the logical unit is a letter, this letter should correspond to the letter used on the ASG control card that assigns the tape (see EXEC III manual). If the logical unit is 0, 1 or 2 then TAPE will act as if PRINTER, CARDS or PUNCH, respectively, had been used instead. Thus

WRITE(TAPE('0'), <list name>) has exactly the same effect as
WRITE(PRINTER, <list name>) which is the same as
WRITE(<list name>)

Note that if the logical unit is 0, 1 or 2 then <numeric> has no meaning and is not examined.

DRUM SIMULATED TAPES

Drum simulated tapes (DST's) enable the user to regard drum as a magnetic tape unit. Such a simulation has the advantage of using an intrinsically faster I/O device and much reduced rewind and position times.

To select a DST, the logical unit should have a value between 4 and 9 inclusive. If a tape length is not specified on the first reference to a DST, then the DST will be made long enough to include all of the allowable drum that is unused by any other DST. The sum of the lengths of all drum tapes must not exceed 524 feet.

Drum simulated tapes are indistinguishable from normal magnetic tape in all respects except for the following:

1. DST's cannot be rewound with interlock (see the section describing the REWIND procedure).
2. DST's do not need to be assigned by an ASG card.
3. Once a length has been assigned to a DST, it will not be changed for the remainder of the program.
4. DST's are erased at the end of each run. Thus a DST must be initialized by writing on it before reading from it.
5. The device DRUM (q.v.) should never be used when DST's are being used.

EXAMPLE

```
REWIND(TAPE('4120'))
```

This procedure call will initialize DST '4' to 120 feet if this DST has not been used before. In any case, DST '4' is rewound.

NOTE: Unless indication is made to the contrary, the rest of the explanation of TAPE applies only to magnetic and drum simulated tape.

DETAILS OF TAPE FORMAT

The smallest unit of information that can be written to tape is a word. Since writing information to tape a word at a time is extremely wasteful of space, information is collected together into 255 word bundles by Algol before being written. Such a bundle will be called a block. An additional word is added to each block by Algol and then the block is written to tape. The extra word is for the device TAPE's own use and is completely undetectable by the Algol programmer.

When reading from or writing to tape, the number of words read or written by each type of Algol variable is given below.

1. An INTEGER, BOOLEAN or REAL datum consists of one word.
2. A REAL2 or COMPLEX datum consists of two words.
3. A STRING datum consists of the number of words given by the formula

   $$1 + ((CH + 5 + MOD(ST - 1,6))//6)$$

   where CH is the number of characters in the desired string and ST is the position of the first character of the desired string in the outermost string.
4. For all types of arrays except string arrays, the number of words composing the array is found by multiplying the number of elements

in the array by the size of each element (see steps 1 and 2).

5. The number of words composing a string array is found by multiplying the number of elements in the array by

$$(CH + 5)//6$$

where CH is the length in characters of each element of the array.

Splitting data between blocks has no significance when writing the tape. If an entire block has not been accumulated when the output parameter list is exhausted then a short block is written consisting of as many words as have been accumulated. The user should be aware that blocks of less than twenty words will generally cause unreliable operation of the tape unit.

OUTPUT TO TAPE

To output data to magnetic and drum simulated tape, the device name in WRITE must be TAPE(<exp>). The other parameters to WRITE (q.v.) have the following specific interpretations.

<label> is utilized in the following fashion:

1. Exit is made to the ERR label if the tape cannot be written properly.
2. Exit is made to the EOI label if either the physical end of tape is sensed or the remaining tape length is indicated to be zero (by utilizing the tape length).

If both labels are missing then exit is made to the Algol error routine instead.

<format> is not allowed as a parameter.

<list name> and <actual parameter list> are the same as described in the previous chapter with the following exceptions:

1. A substring array name without subscripts is not allowed.
2. A class of procedures called modifiers are allowable parameters. These procedures are described in the next section called MODIFIERS.

Note that whenever a tape unit is switched from writing to any other operation, an EOI (end of information) marker is written on the tape following the last word of information. While the user can position past the EOI marker, it is almost always catastrophic to do so.

The actual transmission of data to the tape proceeds concurrently with execution of the program. Consequently, the program may terminate before all the data has been written to tape. To insure that output has terminated correctly the user should do any non-output tape operation on all units that were written to last just before exiting

from the program. An example is rewinding all tapes used for output at the end of the program.

EXAMPLE:

The statement

WRITE(TAPE('G'), DARN, DONEFOR, ARRAY1, ARRAY2)

will cause the arrays named ARRAY1 and ARRAY2 to be written on the tape mounted on logical unit G. If a tape error occurs during the operation, exit will be made to the statement labeled DARN. If the end of the tape is sensed before the write is completed, exit will be made to the statement labeled DONEFOR.

MODIFIERS

The modifiers described in this section may only be called as parameters to WRITE and POSITION. Any call on them in contexts other than these two will cause disastrous (and probably not immediately observable) errors. Only the use of modifiers with WRITE will be discussed in this section, but it should be kept in mind that in both WRITE and POSITION the modifier called is the same procedure and consequently the effect is the same.

The two modifiers allowed in WRITE are KEY and EOF. In the following discussion, the term sentinel will be the generic term used to refer to the class constituted by KEY and EOF.

Sentinels allow a user to partition his data on tape in a manner similar to the way EOF control cards partition a data card deck. Encountering any EOF mark while reading tape causes the same action as encountering an EOF card does when reading cards. KEY marks are ignored during input.

A call on KEY or EOF may be any of the following forms:

1. KEY
2. KEY(<string exp>)
3. KEY(<integer exp>)
4. EOF
5. EOF(<string exp>)
6. EOF(<integer exp>)

The parameters to KEY and EOF serve to distinguish between different EOF marks and between different KEY marks. If the parameter is <string exp> (forms 2 and 5) then only the first five characters are used to identify the sentinel. If less than five characters are specified then blanks are appended to the end. If the parameter is <integer exp> (forms 3 and 6) then only integers in the range

$$-(2^{30}-1) \leq \text{<integer exp>} \leq 2^{30}-1$$

are used to identify the sentinel. If integers outside this range are used

then the result of MOD(ABS(<integer exp>), 2**30) * SIGN (<integer exp>) is used.

No type conversion is made on the parameters to EOF and KEY and if the parameter is not of type STRING or INTEGER then an error message is given. The mark written on tape by each of the above forms is distinguishable from each of the other forms regardless of the parameter supplied. For example all of the following calls produce different sentinel marks.

KEY KEY(123) KEY('123') EOF('123') EOF(123) EOF

When a modifier is encountered in an output parameter list, the following sequence of actions occurs:

1. Output all the parameters occurring before the modifier.
2. Write the appropriate sentinel on tape.
3. Continue processing the output parameter list.

Note that any number of sentinels may be written by a single call on WRITE. Also any number of sentinels may be written consecutively on tape.

The use of modifiers as parameters to POSITION is discussed in the section discussing the POSITION procedure.

INPUT FROM TAPE

To read data from magnetic or drum simulated tape the <device name> in READ must be TAPE(<exp>). The other parameters to READ (q.v.) have the following specific interpretations.

<label> is utilized in the following fashion:

1. Exit is made to the EOF label when an EOF mark is encountered. EOF marks are described in section MODIFIERS.
2. Exit is made to the EOI label when either the end of written information or the physical end of tape is encountered.
3. Exit is made to the ERR label whenever a tape cannot be read properly.

If any of the labels are missing then exit is made to the Algol error routine instead.

<format> is not allowed as a parameter

<list name> and <actual parameter list> are the same as for card input with the exception that a substring array name without subscripts is not allowed.

Only tapes written by calls on the WRITE procedure can be read by the READ procedure. Each call on READ will cause a new block to be read from

tape. Words are extracted from the block as needed and substituted into elements of the input parameter list. If there is not a sufficient number of words in the first block to exhaust the input list, then blocks will continue to be read until enough words have been read. Any information in the last block read that is not used is discarded.

No conversion of type is ever made during tape input. The following discussion indicates how data can be read properly from tape. Define an output list to be that portion of an output parameter list that occurs between two modifiers (excluding the modifiers) such that a) no modifier is included in the output list and b) if the parameter adjacent to either end of the output list is added to the output list then the output list would include a modifier. The parentheses of the WRITE procedure will be considered to act as modifiers in the sense of determining output lists. Also <device name> and <label> will never be part of an output list.

EXAMPLE:

```
LIST    OUT(A, B, ARAY1, KEY, C, D, EOF,EOF, E, ARAY2) $
WRITE(TAPE('A'), LAB1, KEY,OUT, BOOL, 2**10)
```

The output lists in the above example are

```
A,B,ARAY1
C,D
E,ARAY2, BOOL, 2**10
```

An input list will denote a proper input parameter list that is constructed from an output list in the following manner:

1. For every variable and identifier on the output list substitute, respectively, a variable or identifier that has exactly the same arithmetic type as the parameter on the output list. If the output parameter is a string then the input parameter must be a string of exactly the same number of characters. If the output parameter is an array then the input parameter must be an array of exactly the same number of dimensions and the same number of elements per dimension.
2. For every expression and constant not included in step 1 a variable or identifier should be substituted that has the same arithmetic type as the expression or constant.

EXAMPLE:

```
LIST OUT(A,B ARAY1,KEY('2'),C,D,EOF(-1),EOF(3),E,ARAY2) $
WRITE (TAPE('A'), LAB1, KEY(1), OUT, BOOL, 2**10)
```

The following are proper input lists that correspond to the output lists

OUTPUT LIST	INPUT LIST
A,B,ARAY1	INT1, INT2, ARAY
C,D	C,D
E,ARAY2,BOOL,2**10	E,ARAY3,BOOL1,INT

Proper input will occur if the sequence of input lists used in READ is the same as the sequence of output lists used the WRITE that created the tape. Also if the tape is positioned to a point just after a modifier, then the sequence of input lists can start with the list that corresponds to the output list that followed that modifier. Note that no detection of a KEY mark will be possible during input. However, any attempt to read past an EOF mark will cause an exit to the EOF label.

EXAMPLE

In relation to the above example the following are correct sequences of input lists

INT1,INT2,ARAY

INT1,INT2,ARAY,C

C,D,E,ARAY3 if the tape was positioned to KEY(2) first.

Only C and D will be read into since an EOF mark (EOF(-1)) will be encountered when attempt is made to read into E.

E,ARAY3,BOOL1,INT if the tape was positioned to EOF(3) first.

Correct calls on the READ procedure corresponding to the above might be

READ(TAPE('A'),INT1,INT2,ARAY)

READ(TAPE('A'), INT1,INT2,ARAY,C)

READ(TAPE('A'),LAB3,C,D,E,ARAY3)where LAB3 is the EOF label.

READ(TAPE('A'),E,ARAY3,BOOL1,INT)

assuming the tape is positioned properly first.

Since input from tape is not checked to see if it is the proper arithmetic type, a great deal of flexibility (and responsibility) is allowed the user. Data is substituted into input parameters on the basis of the number of words each parameter occupies (see DETAILS OF TAPE FORMAT). For instance, a two dimensional array can be written to tape in one call on WRITE and read back a row at a time into a vector. There are two special cases of which the user must be cognizant. Precautions must be taken when inputting strings and string arrays. No check is made when reading these two types of data so that if the following rules are not observed, only the user will experience difficulty.

When a string is written to tape, it is prefaced with a control word that is interpreted by Algol. To correctly read in a string the tape must be positioned so that the control word will be the first word read. Also the string input parameter must have exactly the same number of characters as the string on tape.

A string array is not written to tape in the same format as a string. Consequently, string arrays must be read in the same manner as they were written. The string array input parameter should have exactly the same number of dimensions, the same number of elements per dimension for each dimension and the same number of characters per element as the string array that was originally written on tape.

THE POSITION PROCEDURE

The POSITION procedure is a fully recursive procedure used to position a magnetic or drum simulated tape. The general form of POSITION is:

POSITION(<label>,<position pair>)

These two parameters may occur in any order.

<label> consists of one or two labels (separated by a comma if two) which are used as alternative exits by the POSITION procedure. These exits are utilized when abnormal conditions are encountered while positioning a tape. The appropriate exit is chosen in the following manner:

i) Exit is made to the first label (if present) if either the beginning or the end of written information is encountered before satisfying the position. This label is called the EOI label.

ii) Exit is made to the second label (if present) if the tape cannot be moved because of some malfunction of the tape unit. This label is called the ERR label.

<label> need not be specified. The standard Algol error exit is supplied for all missing labels. If an exit is made to the Algol error routine, an appropriate message is printed and the program is terminated.

<position pair> consists of a call on TAPE, a comma and a position modifier. A position modifier may be any one of the following eleven forms:

1. KEY or +KEY
2. EOF or +EOF
3. -KEY

4. -EOF
5. EOI or +EOI
6. -EOI
7. <integer exp>
8. KEY (<exp>) or +KEY (<exp>)
9. EOF (<exp>) or +EOF (<exp>)
10. -KEY (<exp>)
11. -EOF (<exp>)

<exp> may be either a string expression or an integer expression. In either case it has the same effect as a write modifier. In all eleven forms if the position modifier is unsigned or positive then the tape is moved in the forward direction; otherwise it is moved in the backward direction.

If position modifier is a KEY or EOF (forms 1,2,3,4,8,9,10 and 11) then the designated tape unit is moved in the appropriate direction until the closest specified KEY or EOF mark is found. If no such KEY or EOF mark is found then exit will be made to the EOI label if it is present. If the position is in the forward direction then the tape will be positioned immediately following the KEY or EOF mark. If the position is in the backward direction then the tape will be positioned immediately preceding the KEY or EOF mark. In this case an attempt to read forward will exit through the EOF label if present if it is an EOF mark.

If position modifier is an EOI (forms 5 and 6) then the tape is moved to the beginning or end of information, respectively. If it is a position to the end of information the tape is left so that a call on WRITE will remove the EOI marker from that spot on the tape.

If position modifier is an integer expression then the tape is positioned <integer exp> blocks in the appropriate direction. If the end of written information is encountered before the requisite number of blocks have been positioned, then exit will be made to the EOI label if present.

EXAMPLE:

```
POSITION(TAPE('D'),KEY('SIS'),LAB1,LAB2)
```

This statement will initiate a forward search of the tape on logical unit D for a KEY mark having as identifier the string 'SIS'. If such a KEY mark is not found, control resumes with the statement labeled LAB2.

THE REWIND PROCEDURE

The REWIND procedure is a fully recursive procedure used to rewind

tape units. The general form of the REWIND is:

REWIND(<tape list 1>, <modifier>, <tape list 2>)

where any combination of the parameters may be omitted.

<tape list 1> and <tape list 2> are calls on TAPE separated by commas.

EXAMPLE:

TAPE('A'),TAPE('P1200'),TAPE(S(1)),TAPE('4')

where S is a string.

For REWIND, logical units of 0, 1 and 2 are permitted. If any of these logical units occur, they are ignored. All tape units specified by <tape list 1> are rewound. <tape list 2> is explained below.

<modifier> is the procedure call INTERLOCK. If INTERLOCK occurs in the parameter list, then all tapes designated by <tape list 2> are rewound with interlock. All tapes in <tape list 1> are unaffected.

EXAMPLE:

REWIND(TAPE('4'),TAPE('1'),TAPE('P'),INTERLOCK,TAPE('P'))

Here drum simulated tape 4 is rewound, and logical unit P is first rewound and then rewound with interlock. Rewinding logical unit 1 produces no action.

Drum simulated tapes cannot be rewound with interlock. If such an operation is attempted then the tape is rewound.

THE DRUM DEVICE

DRUM is a non-recursive procedure that specifies a <device name> to the I/O procedures READ and WRITE. When DRUM is used as a parameter to either of these procedures it must have one parameter of type INTEGER. The parameter specifies an address on drum at which an input or output operation is to start. All drum operations will proceed from this starting address to increasing addresses. The starting address will have an implied range of allowable values which is

$$0 \leq \text{<starting address>} \leq 262{,}143.$$

This range can be doubled by making the declaration

EXTERNAL SLEUTH PROCEDURE DRUM $

When the declaration is in effect, there will be no automatic type conversion of the parameter of DRUM to type integer. If such a conversion is necessary it must be explicitly made by the user. This declaration has the following effects:

1. The starting address now has the allowable range

   $$0 \leq \langle\text{starting address}\rangle \leq 524{,}287$$

2. What used to be address 0 in the implied range becomes address 262,144 and so on for all higher addresses.
3. Any starting address between 0 and 262,143, inclusive, will input or output from the users' program complex file (PCF).

The device DRUM should never be used when drum simulated tapes are being used since the same area of drum is used by both. The amount of drum used by drum simulated tapes remains fixed and cannot be altered by the above declaration of DRUM.

The starting address indicates where input or output is to begin. Each address corresponds to a word of data on drum. If more than one word is needed for input or output then consecutive higher addresses are used. Data is represented on drum exactly as it is on tape except, of course, that there are no blocks on drum. Also, there is no special internal control word for every 255 words of data. The number of words occupied by each kind of Algol variable is given below.

1. An INTEGER, BOOLEAN or REAL datum consists of one word.
2. A REAL2 or COMPLEX datum consists of two words.
3. A STRING datum consists of the number of words given by the formula

   $$1 + ((CH + 5 + \text{MOD}(ST - 1, 6))//6)$$

   where CH is the number of characters in the desired string and ST is the position of the first character of the desired string in the outermost string.
4. For all types of arrays except string arrays, the number of words composing the array is found by multiplying the number of elements in the array by the size of each element (see steps 1 and 2).
5. The number of words composing a string array is found by multiplying the number of elements in the array by

   $$(CH + 5)//6$$

   where CH is the length in characters of each element of the array.

Note that the final address written into or read from must be in the same range as the starting address.

The user should be aware that there is no predictable data on drum at the start of his program. It is necessary for the programmer to initialize the drum by writing on it before anything is read from drum. Also all data is erased at the end of each run.

DRUM AS A PARAMETER TO READ

When DRUM is specified as the device in a READ procedure call, the labels that are parameters are interpreted in the following fashion:

i) The EOF label is never used.

ii) Exit is made to the EOI label when an attempt is made to read past address 262,143 (or 524,287 of enlarged drum).

iii) Exit is made to the ERR label if the drum cannot be read correctly.

<format> is not allowed as a parameter.

<list name> and <actual parameter list> are the same as for card input with the exception that a substring array name without subscripts is not allowed.

EXAMPLE:

The statement

READ(DRUM(MAX(THIS,THAT)),PROJECTS,RECOVER)

will read the drum starting at the address given by whichever INTEGER variable has the larger value, THIS or THAT. The array PROJECTS will be filled by READ and the error label is RECOVER.

DRUM AS A PARAMETER TO WRITE

When DRUM is specified as the device in a WRITE procedure call, the label parameters are interpreted in the following way.

1. Exit is made to the ERR label if the drum cannot be written properly.
2. Exit is made to the EOI label when an attempt is made to write past address 262,143 (or 524,287 if the larger drum area is in use).

<format> is not allowed as a parameter.

<list name> and <actual parameter list> are the same as described in the previous chapter with the exception that a substring array name without subscripts is not allowed as a parameter.

EXAMPLE:

WRITE(DRUM(I),ARAY)

This procedure call will write array ARAY to drum starting at address I.

SPEED OF DRUM AND TAPE INPUT/OUTPUT

The following suggestions are made to allow the user to maximize the speed of input/output operations to tape or drum. These suggestion do not need to be observed.

It is always faster to specify an array by name only instead of specifying each element separately.

Input of a string will be faster if, when the string is written out, the position of the first character of the string has character position 6*N+1 (for some N) in the outermost string and the string that is the input parameter satisfies the same restriction. If both of these conditions are met, 6*((CH + 5)//6) (where CH is the number of characters in the input parameter string) characters will always be read.

LIBRARY PROCEDURES FOR TAPE AND DRUM I/O

Procedure Call	Type of Argument(s)	Type of Result	Description
DRUM(<exp>)	INTEGER, REAL, REAL2, STRING	none	Sets starting address for drum Input/Output operation equal to <exp>. If EXTERNAL SLEUTH PROCEDURE DRUM $ has been used then the parameter to DRUM must be of INTEGER type.
±EOF(<exp>)	STRING or INTEGER	none	Used as parameter to POSITION and WRITE. Positions to designated EOF mark on selected tape unit when used as parameter to POSITION. Writes EOF mark when used as parameter to WRITE.
±EOI	none	none	Used as parameter to POSITION. Will position to either beginning or end of written information depending on sign.
INTERLOCK	none	none	Used with REWIND to rewind tape units with interlock
±KEY(<exp>)	STRING or INTEGER	none	Used as parameter to WRITE and POSITION. Positions to designated KEY mark on selected tape unit when used as parameter to POSITION. Writes KEY mark when used as parameter to WRITE.
TAPE(<exp>)	STRING	none	Defines tape unit specified by <exp> as device in Input/Output procedure

XI...

BLOCKS

BLOCKS

In Algol 60, a program consists of <u>one</u> block which in turn may contain within it many subblocks nested to any depth. These subblocks serve to define a structure within a program which facilitates the construction of a program by permitting it to be built up of pieces -- each of which may be relatively independent of each other. These pieces may in turn be tested separately before putting them together in the final program.

An upper bound of 1,020 different blocks per program is imposed by the Algol compiler not including any external sections (see Chapter XII), This bound should be adequate for most programs, but if necessary this number can be extended by using external procedures. The blocks are numbered from 1 to 1,020 in the order in which the program is read by the compiler.

An example of a program with a complex block structure is given in Figure 2.

BLOCK FORMAT

A block consists of two parts; namely, the heading and the body. The heading may not be empty and the block head must precede the body of the block.

The block head consists of all declarations needed within the block. This forces all identifiers, procedures, lists, formats, arrays, etc. to be defined before they are used.

The body of the block on the other hand contains all statements pertinent to the block which are not declarations. This means that all of

the active statements such as replacement statements must be located in the body of the block.

```
DONOTHING..     BEGIN
                INTEGER  I  $
                REAL ARRAY  A(1..20)  $
                FOR I = 1 STEP 1 UNTIL 20 DO
                        A(I) = 0.0
                END  DONOTHING $
```

In the above example, the declarations INTEGER $ and REAL ARRAY A(1..20) $ constitute the head of the block while the FOR statement and replacement statement form the body of the block.

DEFINING A BLOCK

A block is defined by enclosing a section of program consisting of a head and body within a set of BEGIN - END statement parentheses, as in the above example. This, in effect, means that the word BEGIN followed immediately by some form of declarations defines a new block which is terminated by the END that matches the defining BEGIN. The double use of the BEGIN - END parentheses should be noted by the user. A group of statements enclosed by a BEGIN - END pair will form a compound statement if the statement following the BEGIN is not a declaration. On the other hand if this statement is a declaration then a new block will be defined rather than a compound statement.

A special rule has been incorporated in the compiler for defining the outermost block in a program (block 1 in figure 2). The normally required BEGIN - END pair is not required in this instance as the compiler will automatically supply a matching BEGIN - END pair. However, if the user desires to supply his own BEGIN - END as the formal language requires, no error will result in the compiled program. All inner blocks do require the bounding BEGIN - END pair.

EXAMPLES:

Program 1

```
COMMENT         THIS PROGRAM INVERTS A SQUARE MATRIX
                BY STRAIGHT GAUSS ELIMINATION.
                A = MATRIX, N = ORDER OF A.
```

```
INVERSE OF A IS WRITTEN OVER A $
INTEGER N $
READ (N) $
      BEGIN REAL ARRAY A(1..N, 1..N), V(1..N) $
      INTEGER I, J, C $
      READ (A) $
      FOR C = 1 STEP 1 UNTIL N DO
            BEGIN FOR I = 1 STEP 1 UNTIL N-1 DO
            V(I) = A(1, I+1)/A(1,1) $
            V(N) = 1/A(1,1) $
            FOR I = 1 STEP 1 UNTIL N-1 DO
                  BEGIN FOR J = 1 STEP 1 UNTIL N-1 DO
                  A(I,J) = A(I+1, J+1) - A(I+1, 1)*V(J) $
                  A(I,N) = -A(I+1, 1)*V(N)
                  END $
            FOR J = 1 STEP 1 UNTIL N DO A(N,J) = V(J)
            END $
      WRITE ('INVERSE OF A',  A)
      END
```

Program 2

```
COMMENT     THIS PROGRAM DETERMINES THE SPECTRAL RADIUS OF A SQUARE
            MATRIX BY THE POWER METHOD, A = MATRIX, N = ORDER OF A $
            INTEGER N $ READ (N) $
            WRITE ('ORDER OF A = ',N) $
                  BEGIN INTEGER I,J,K $
                  REAL MAXQ, MINQ, SUM, YMAX $
                  REAL ARRAY A(1..N,1..N), X,Y(1..N) $
                  LIST QUOTIENTS(FOR K = 1 STEP 1 UNTIL N DO ABS(Y(K)/X(K))) $
                  LIST YABS(FOR K = 1 STEP 1 UNTIL N DO ABS(Y(K))) $
                  READ (A) $ WRITE ('MATRIX A',A) $
                  FOR I = 1 STEP 1 UNTIL N DO X(I) = 1.0 $
REPEAT..          FOR I = 1 STEP 1 UNTIL N DO
                        BEGIN SUM = 0.0 $
                        FOR J = 1 STEP 1 UNTIL N DO
```

```
                SUM = SUM + A(I,J) * X(J) $
                Y(I) = SUM
                END $
            MAXQ = MAX(QUOTIENTS) $
            MINQ = MIN(QUOTIENTS) $
            IF (MAXQ - MINQ)/MAXQ LSS &-5 THEN
                BEGIN WRITE ('SPECTRAL RADIUS OF A',MAXQ) $
                GOTO FINA END $
            YMAX = MAX(YABS) $
            FOR I = 1 STEP 1 UNTIL N DO X(I) = Y(I)/YMAX $
            GOTO REPEAT $
FINA..      END
```

LOCAL AND GLOBAL IDENTIFIERS

All identifiers that are declared explicitly within a given block are said to be local to that given block. Any identifiers that are defined in an outer block of the given block (i.e. in any block that contains the given block) and are not redefined in the given block are said to be global to the given block. Also any identifiers that are defined within an inner block (i.e. any block contained in the given block) will have no meaning in the given block and cannot be referred to from the given block.

Since a label is an identifier, this last statement means that all blocks are entered through their heads and it is impossible within the language to enter the middle of a block. Also all variables that are defined in a block and hence local to the block will be bound to the block.

When a block is entered (through its head) space will be requested and taken away from the remaining available space in memory and assigned to all normally defined local variables in the block including arrays. The initial values of these variables are to be considered as undefined, except for OWN variables. The value zero is assigned to OWN variables of type INTEGER, REAL, REAL2, and COMPLEX while OWN STRING's are set to blanks and OWN BOOLEAN variables are initialized to FALSE. When an exit of any form, whether by means of a GOTO statement or normally through the end of a block is made from a block, all memory space that is assigned to local normal variable storage is returned to the available

space part of memory and can be used by other blocks for local variable storage, At times it is desirable to have variables retain their values from block entry to block entry. Since normal variables will have their values redefined to zero upon each entry and thrown away on each exit, a special class of variables called OWN variables is defined. This class has permanently allotted space in memory and variables named in an OWN declaration will retain their values from entry to entry. However, it should be noted that the identifier rule still holds, and these variables although residing in memory cannot be addressed from outside the defining block. See Chapter V, THE OWN DECLARATION, for a description of storage assignment of Block 1 quantities.

To further illustrate the concept of local and global identifiers consider the block displayed in Figure 1. The variables I, J, K, X, Y, Z and labelsL1, L2, are local identifiers in Block 1. Only I, K, X, Y, Z, L1, L2 will be global to Block 2, while J is redefined and local in Block 2 along with L, M, U, V. In Block 3, I, Y, Z, L1, L2 from Block 1 are global along with J, L, U, V from Block 2. K, M, N, W, X are local in Block 3. Consider the statement, L3. The global variable Y will be replaced by the sum of the local variable X with the product of the global variables V, Z. The statement L4 in Block 4 looks the same as L3 in Block 3. However, the variable X in this case refers to the variable X in Block 1 rather than the varialbe X in Block 3; hence the effect of the statements will be different.

In Block 5, the labeled statement L5 has an erroneous GO TO L4 after the word THEN. The label L4 is defined in Block 4 and has no meaning within Block 5 since the blocks are disjoint. However, the GO TO L1 is correct and will send control to the statement labeled L1 in Block 1. This in effect will cause the program to re-enter Block 2.

Figure 2

```
INTEGER I,J,K $                                    (Block 1)
REAL X,Y,Z $
L1..BEGIN
        INTEGER J,L,M$                             (Block 2)
        REAL U,V$
        BEGIN                                      (Block 3)
                INTEGER K,M,N $
                REAL W,X $
                L3..Y=X+V*Z $

        END $

        BEGIN                                      (Block 4)
                BOOLEAN B1, B2 $
                INTEGER P, Q $
                REAL Z1, Z2 $
                L4..Y=X+V*Z $

        END
END $

L2..BEGIN                                          (Block 5)
        COMMENT  NOTE THAT THE STATEMENT LABELLED L5 IS IN ERROR  $
        INTEGER V, Z $
        REAL I, M $
        BOOLEAN B1$
        L5..IF B1 THEN GOTO L4 ELSE
        GOTO L1 $

END
```

XII...

PROCEDURES

A procedure in Algol is a very general and flexible structure which includes as a subset the more or less generally recognized classes of subroutines and functions. It is generally used to specify a section of program, which usually represents an algorithm, as a somewhat independent piece of the program that can be called or reused many times from other parts of the program. This structure enables one to test various independent pieces of a program before putting them together in a more complicated way.

The procedure may be classified or subdivided into three categories, namely: normal Algol procedures (simply called procedures), library procedures, and external procedures. First, let us consider the class of normal Algol procedures.

These procedures represent a special type of block called a procedure block. Procedure blocks are defined by means of a procedure declaration and exhibit most of the normal properties of blocks.

THE PROCEDURE BLOCK

A procedure block consists of a procedure heading followed by either a standard block or a statement, which represents the body of the procedure. The procedure heading consists of three parts, namely: the procedure declaration with formal parameter list, the value part, and the specification part.

THE PROCEDURE DECLARATION

The format of the procedure declaration is as follows:

<type> PROCEDURE <identifier>
<type> PROCEDURE <identifier> (<par$_1$>, ..., <par$_N$>)
<type> PROCEDURE <identifier> (<par$_1$>) <str$_2$>: (<par$_2$>)
<str$_3$>: ... (<par$_N$>)

where <identifier> is the name of the procedure, the <par$_I$> are the formal parameters of the procedure, the <str$_I$> are strings which represent parenthetical comments (and need not be included), and <type> (see Chapter V) is one of the following: REAL, REAL2, INTEGER, BOOLEAN, COMPLEX and STRING. It should be noted that <type> may be empty. However, if <type> is empty, the procedure cannot be used in the functional sense (see below). Since the first parameter must be written immediately after the procedure name, the maximum number of comment strings is one less than the number of parameters.

When the procedure declaration is encountered at run time, space is created for each of the formal parameters in a fashion similar to that used for local variables in a block. The formal parameters are considered as local identifiers to the defining procedure block, and may be used in the global sense in any block nested inside the procedure block. When the procedure is called, space is allotted for all local variables and parameters, and all of the actual parameters are moved into the cells allotted to the formal parameters.

The procedure declaration must be present in all procedure blocks and is the first part of the procedure heading.

EXAMPLES:

```
PROCEDURE INVERT (A, B) $
REAL PROCEDURE SIMPS (A,B) FUNCTION.. (F) ERROR .. (DELTA) $
```

Note that

)<str$_I$>:(

is equivalent to a comma as a separator of parameters.

THE VALUE PART

This part of the procedure heading is optional and has the following format:

GENERAL FORM:

VALUE <formal parameter list> $

The <formal parameter list> consists of those formal parameter identifiers contained in the procedure declaration, which are to be considered as values, separated by commas. These parameter values are obtained from the procedure call when the procedure block is entered and remain fixed (unless altered by a replacement statement) throughout the body of the procedure. Any formal parameter occurring in the VALUE part is considered to be a call-by-value parameter or simply a value parameter. A value parameter behaves identically to a local variable except that its initial value is obtained from the procedure call rather than being considered undefined.

It should be noted that a value parameter which is an array identifier will cause the entire array supplied by the procedure call to be copied locally within the procedure block. This, in effect, may cause large amounts of memory space to be unexpectedly used when the procedure is called. Also, value parameters that correspond to labels will cause any switch variable or designational expression in the call to be evaluated upon entry, as expected.

If the arithmetic type of the actual parameters in the procedure call differ from those of the formal parameters, then an appropriate type conversion will be performed (if possible) for those cases in which the formal parameter is a value parameter. All other cases will result in an error message at run-time.

The value part (if present) must follow the procedure declaration and precede the specification part.

EXAMPLE:

VALUE X, A, Z $

THE SPECIFICATION PART

All formal parameters defined by a procedure declaration must be specified with regards to <type> in the specification part of a procedure heading. The format of the specification part is as follows:

GENERAL FORM:

<specification> <formal parameter list>

The <specification> has one of the following forms:

<type>
ARRAY
<type> ARRAY
PROCEDURE
<type> PROCEDURE
LABEL
SWITCH
FORMAT
LIST
<define classification>

and <type> is one of the following;

INTEGER
REAL
REAL2
BOOLEAN
COMPLEX
STRING

The <formal parameter list> again consists of the formal parameter identifiers contained in the procedure declaration separated by commas.

The reason that all formal parameters must be specified is that the compiler must know the type of all parameters in order to compile proper machine code.

EXAMPLES:

```
INTEGER I, K $
REAL X, Y $
REAL ARRAY Z $
BOOLEAN PROCEDURE F $
STRING S $
```

The details of constructing a procedure block can best be described by displaying several examples.

EXAMPLES:

```
PROCEDURE  SPUR(A) ORDER..(N) RESULT..(S)  $
```

```
        VALUE  N $ ARRAY  A  $  INTEGER  N  $  REAL  S  $
COMMENT PROCEDURE SPUR COMPUTES THE TRACE OF MATRIX A  $
        BEGIN  INTEGER  K  $
        S = 0.0 $
        FOR K = 1 STEP 1 UNTIL N DO S = S + A(K,K)
        END   SPUR $


        PROCEDURE TRANSPOSE (A,N) $
        VALUE N $ ARRAY A $ INTEGER N $
COMMENT    PROCEDURE TRANSPOSE REPLACES MATRIX A BY THE TRANSPOSE OF A,
           WHICH IS A SQUARE MATRIX OF ORDER N $
        BEGIN  REAL W $   INTEGER  I, K $
        FOR I = 1 STEP 1 UNTIL N DO
        FOR K = I+1 STEP 1 UNTIL N DO
                BEGIN  W = A(I,K)  $
                A(I,K) = A(K,I)  $
                A(K,I) = W
                END
        END  TRANSPOSE $

        INTEGER PROCEDURE STEPS(U) $ REAL  U$
        STEPS = IF 0 LEQ U AND U LEQ 1 THEN 1 ELSE 0

        PROCEDURE ABSMAX(A)  SIZE..(N,M)  RESULT..(Y) SUBSCRIPTS..(I,K)  $
        VALUE N, M $
        ARRAY  A  $  INTEGER  N, M, I, K $  REAL  Y $
COMMENT   PROCEDURE ABSMAX ASSIGNS TO Y THE MAGNITUDE OF THE ELEMENT OF
          GREATEST ABSOLUTE VALUE IN MATRIX A, AND ASSIGNS TO I AND K
          THE SUBSCRIPTS OF THAT ELEMENT  $
        BEGIN INTEGER  P, Q  $
        Y = 0.0  $
        FOR P = 1 STEP 1 UNTIL N DO
        FOR Q = 1 STEP 1 UNTIL M DO
        IF ABS(A(P,Q)) GTR Y THEN
              BEGIN  Y = ABS(A(P,Q))  $
              I = P  $  K = Q
```

```
    END
END  ABSMAX  $
```

```
    PROCEDURE INNERPRODUCT (A,B)  ORDER..(K,P)  RESULT..(Y) $
COMMENT   PROCEDURE INNERPRODUCT AND (MODIFIED) CALL
          TAKEN FROM REVISED ALGOL 60 REPORT $
    VALUE K $  INTEGER K, P $  REAL Y, A, B $
    BEGIN  REAL S $
    S = 0.0 $
    FOR P = 1 STEP 1 UNTIL K DO S = S + A*B $
    Y = S
    END INNERPRODUCT  $
```

VALUE AND NAME PARAMETERS

All formal parameters that have been listed in a VALUE part are called value parameters. These parameters will be assigned a value corresponding to the value of the actual parameter in the procedure call upon entering the procedure block. Any changes made in the value parameters within the body of the procedure block will have no effect outside the body of the procedure.

Any formal parameters that have not been listed in a VALUE part are called name parameters. The address of the actual parameter in the procedure call will be assigned to the formal parameter upon entering the procedure block. Consequently, any changes made in the name parameter within the body of the procedure block will be carried outside to the actual parameters supplied by the calling block.

This property of name parameters permits a large number of results to be supplied to the calling program from within the procedure. One word of caution: This property can sometimes cause far-reaching and disastrous effects in the overall program by altering either the values of variables in the calling block or the actual parameters in the call when least expected. (Consider in detail the GPS procedure below).

FUNCTIONAL PROCEDURES

Procedures which are to be used in the functional sense (e.g. SIN, EXP) must have a <type> associated with the procedure identifier (i.e. procedure name). This <type> declaration must be the first symbol of the procedure declaration. Also for the functional procedure to have a value associated with it, the procedure identifier must occur at least once as the left part of an assignment statement in the procedure body. In addition, at least one of these assignment statements must be executed on a given procedure call for a value to be assigned to the procedure. If more than one such assignment statement is executed within the body, then the last one executed before exiting from the procedure determines the value associated with the procedure. Any other occurrences of the procedure identifier within the body of the procedure will be considered as (recursive) calls on the procedure.
EXAMPLE:

```
REAL PROCEDURE FACTORIAL1(N) $
VALUE N $  INTEGER  N $
BEGIN  REAL S  $
INTEGER  I  $
S = 1.0  $
FOR I = 1 STEP 1 UNTIL N DO S = S*I $
FACTORIAL1 = S
END  FACTORIAL1  $
```

THE PROCEDURE CALL

A procedure call has the following format:

<identifier> (<par$_1$>, ..., <par$_N$>)

where <identifier> is the name of the called procedure and the <par$_I$> are the actual parameters supplied to the procedure.

The correspondence between the actual parameters in the procedure call and the formal parameters of the procedure heading is established as follows: The actual parameter list of procedure call must have the same number of entries as the formal parameter list of the procedure declaration heading. The correspondence is obtained by taking the entries of these two lists in the same order. It is the responsibility of the programmer to match the types of parameters in the lists.

A formal parameter which occurs as the left part variable of an assignment statement within the procedure body and which is not called by value (see above) should only correspond to an actual parameter which is a variable. Also, a formal parameter which is used within the procedure body as an array identifier should only correspond to an actual parameter which is an array identifier of an array of the same dimensions and subscript bounds. In addition, if the formal array parameter is called by value, the local array created during the call will have the same subscript bounds as the actual array. Correspondingly, a formal parameter which is used within the procedure body as a string identifier can only correspond to an actual parameter which is a string identifier unless the formal parameter is called by value. In this case, appropriate conversions of type will be made if possible.

The following rule-of-thumb will help avoid problems involving the correspondence between formal and actual parameters: all parameters to a procedure except arrays should be called by value unless there is an explicit reason for doing otherwise. In particular, actual parameters that are constants and procedure calls should correspond to formal parameters that are called by value.

COPY RULE

The procedure call acts as if the call were replaced by the statements in the procedure body with the following changes being made in the body:

1) All formal parameters quoted in the value part of the procedure declaration are assigned values of the corresponding actual parameters, these assignments being considered as being performed explicitly before entering the procedure body.

2) Any formal parameter <u>not</u> quoted in the value list is replaced, throughout the procedure body, by the corresponding actual parameter, after enclosing the actual parameter in parentheses whenever syntactically possible. Possible conflicts between identifiers inserted through this process and other identifiers already present within the procedure body will be avoided by suitable systematic changes of the formal or local identifiers involved.

3) Any global identifier referenced within the procedure has its

identity defined as at the point of declaration, while its value is that in force at the time of the call.

4) The procedure body, as modified above, is executed in place of the call.

RECURSIVE PROCEDURE CALLS

The usual context for a recursive procedure call is an explicitly stated procedure call from within the body of the procedure itself. As an example, consider the following recursive form for the previous factorial procedure:

EXAMPLE:

```
REAL PROCEDURE FACTORIAL2(N)  $
VALUE N  $    INTEGER N    $
FACTORIAL2 = IF N EQL 0 THEN 1.0 ELSE N*FACTORIAL2(N-1)
```

A more obscure method for a procedure to be called recursively is to pass a call for the procedure into the procedure as a name parameter. As an example, consider the following procedure for integration using Simpson's Rule (Frank Olynyk, Comm. of ACM, Vol. 7, June 1964, p.348).

EXAMPLE:

```
REAL PROCEDURE SIMPS (X, X1, X2, DELTA, F)  $
VALUE X1, X2, DELTA  $
REAL  X, X1, X2, DELTA, F  $
      BEGIN
      BOOLEAN TURING $  REAL  Z1, Z2, Z3, H, K  $
      TURING = FALSE  $
      IF X1 EQL X2 THEN
      BEGIN Z1 = 0  $
      GOTO BOX2  END $
IF X1 GTR X2 THEN
      BEGIN H = X1    $    X1 = X2    $    X2 = H  $
      TURING = TRUE   END   $
X = X1   $   Z1 = F  $  X = X2  $  Z3 = Z1 = Z1 + F  $
K = X2 - X1  $
BOX..  Z2 = 0  $  H = K/2  $
```

```
            FOR X = X1 + H STEP K UNTIL XP DO Z   Z2 + F  $
            Z1 = Z1 + 4*Z2  $
            IF H*ABS((Z1-2*Z3)/(IF Z1 E.. 0 THEN ' ELSE Z1))
                  .GE. DELTA THEN GOTO BOX2 ELSE Z3 = Z1  $
            Z1 = Z1 - 2*Z2  $
            K = H  $  GOTO BOX  $
BOX2..      IF TOPING THEN H = -H  $
            SIMPS = H*Z1/3
            END   SIMPS $
```

Using this procedure, the double integral

$$\int_0^1 \int^{(1-X^2)^{\frac{1}{2}}} (X + Y)\, dXdY$$

would be evaluated by the recursive call:

```
ITERINT = SIMPS (X, 0, 1, DELTA,
                 SIMPS(Y, 0, SQRT(1-X**2), DELTA2, X+Y))
```

GENERAL PROBLEM SOLVER

Now for those of the readers who feel that they have mastered the concept of procedures in Algol and who consider themselves sophisticated programmers, consider the following procedure, called GPS for General Problem Solver (D. E. Knuth, Comm. of ACM, Vol. 4, June 1961, p. 271):

EXAMPLE:

```
            REAL PROCEDURE  GPS(I, N, Z, V)  $
            REAL  I, N, Z, V  $
                  BEGIN
                  FOR I = 1 STEP 1 UNTIL N DO Z = V  $
                  GPS = 1
                  END   GPS $
```

Isn't that the most harmless looking procedure you ever saw? Wait a minute, there is a lot of danger as well as opportunity lurking in the call-by-name parameters.

If we wish to calculate the innerproduct of the N-element vectors A and B, we simply write:

```
Z = 0 $  I = GPS (I, N, Z, Z + A(I)*B(I))
```

But we can do much better than that. Suppose we want to multiply the array A(1..M, 1..N) by B(1..N, 1..P) and store the result in C(1..M, 1..P). This can also be done using GPS, by writing

```
I = GPS (I, 1.0, C(1,1), 0.0)*GPS(I, (M-1)*
        GPS(J, (P-1)*GPS(K, N, C(I,J),
        C(I,J) + A(I,K)*B(K,J)), C(I,J+1), 0.0),
        C(I+1, 1), 0.0)
```

Problems which are unrelated to matrix multiplication can also be done with GPS. In fact, we can actually compute any computable function using a single Algol assignment statement containing one or more calls on GPS.

LIBRARY PROCEDURES

Chapter VIII is the basic reference for library procedures. Other material on library procedures will be found in Chapters IX, X and XIII.

EXTERNAL PROCEDURES

External procedures in ALGOL can be coded in any of three ways, namely:

1) Another ALGOL procedure compiled separately.
2) A procedure coded in SLEUTH following specified rules.
3) A FORTRAN subroutine compiled separately.

The coding rules vary depending upon the choice of coding methods. Each method in turn also specifies the form of the procedure declaration within the main ALGOL program.

EXTERNAL PROCEDURE DECLARATION

If the external procedure has been coded in ALGOL and compiled separately, then the name of the procedure must be defined in the main program by means

of an EXTERNAL procedure declaration. The form of the declaration in this case is

EXTERNAL <type> PROCEDURE <proc$_1$>, ..., <proc$_N$> $

where the <proc$_1$> are the names of the desired external procedures, and <type> is non-empty if the procedures are to be used in the functional sense (see Chapter V, DECLARATIONS OF TYPE, for a list of the available types). Omission of the type specification indicates that no functional results will be associated with the names of the procedures.

A procedure declared this way will always be called in the recursive sense by the calling program. It should be noted that only procedures of this form (i.e., recursive) can be passed on to other procedures as actual parameters.

An external procedure may also be written in 1107 Sleuth. In this case the form of the external declaration in the calling program is:

EXTERNAL SLEUTH <type> PROCEDURE <proc$_1$>, ..., <proc$_N$> $

where <proc$_1$> and <type> have the same meanings as in the previous case. Since the details of coding a Sleuth routine that may be called recursively are very complicated, it is assumed that a procedure declared in this manner in non-recursive. Therefore such a procedure cannot be passed to other procedures as parameters and all expressions passed on to the Sleuth procedure will be evaluated before entry to the procedure.

See Appendix II for an explanation of the calling sequence generated for a Sleuth procedure and description of how to handle the parameters in the Sleuth routine.

If the procedure is a function or subroutine written in Fortran, it must be compiled separately and the form of the external declaration in the calling program becomes:

EXTERNAL FORTRAN <type> PROCEDURE <proc$_1$>, ..., <proc$_N$> $

where <proc$_I$> and <type> have the same meanings as previously. (Note that STRING is not acceptable as a <type> for a Fortran function.) A procedure declared in this form is non-recursive and therefore cannot be passed as a parameter to another procedure.

EXTERNAL PROCEDURE CALLS

The form of the procedure call for external procedures is the same as that for internally defined or library procedures, namely

<identifier>(<par$_1$>,...,<par$_N$>)

where <identifier> is the name of the procedure and the <par$_I$> are the actual parameters to the procedure.

If an array is to be passed as a parameter to a Fortran subprogram then the actual parameter should be the first element of the array. For example if the array is two-dimensional and is named SECTION then the actual argument on the call should be SECTION (1,1).

EXTERNAL REFERENCES

To facilitate communication between programs and their subprograms, and to save allocation time, variables, and formats may be externally defined. This is accomplished via the OWN declaration.

All OWN variables and formats declared in block 1 of an Algol program are automatically externally defined. These elements may then be referenced from Algol procedures that are called by the program in question. In the case of variables the procedure must include an external declaration of the form

EXTERNAL <type> <var$_1$>, ..., <var$_N$> $

where <type> is as described in Chapter V, DECLARATIONS OF TYPE, and the <var$_I$> are the variables in question.

In the case of formats the declaration is

EXTERNAL FORMAT <format$_1$>, ..., <format$_N$> $

EXAMPLE:

```
7/8  ALG  PROGRAM

     OWN INTEGER  I, J, K  $

     OWN FORMAT  LINE (I4, A1)  $

     EXTERNAL PROCEDURE  OUTSIDE  $

     I = 1    $    J = 2    $

     OUTSIDE
```

```
7/8E  ALG   PROCEDURE

      EXTERNAL FORMAT  LINE  $

      EXTERNAL INTEGER  I, J, K  $

      PROCEDURE  OUTSIDE  $
      BEGIN
      K = I + J   $

      WRITE (LINE, K)  $
      END
```

Note the following rules when compiling external Algol procedures:

1) the Algol processor card governing the compilation must carry the E option;
2) a program under this option cannot be executed as a mainline program;
3) the E option causes all procedures declared in block 1 to be externally defined;
4) any number of external procedures may be declared in one program;
5) the object programs for all external procedures must be available in the user's program complex on drum before the main program is executed ;
6) variables declared in block 1 of a program compiled under the "E" option can only be simple variables (not including strings). Lists declared in block 1 may contain only such variables;
7) the first six characters of all externally defined identifiers must be unique.

Rules for writing and processing Fortran and Sleuth routines are available in the manuals on those languages. See Appendix II for further information on the linkage between calling program and procedures.

XIII...

THE DIAGNOSTIC SYSTEM

The detection and correction of programming errors, commonly known as debugging, is facilitated in Case Algol by an extensive diagnostic system. Responsibility for error detection is divided between the compiler and the system library routines. By paying close attention to the messages produced by these two segments, the programmer will generally save himself time and aggravation.

As the compiler translates an Algol program it prints the source program instructions and in addition generates two types of information, diagnostics and error messages.

This information is located as close as possible to the instruction to which the message applies and always begins at the extreme lefthand side of the listing.

COMPILER DIAGNOSTICS

Diagnostics are provided by the compiler to help the programmer understand the structure of his program and to call attention to special features that the program uses. Diagnostic messages never inhibit the production of an object program.

The structure of a program is determined by the use of BEGIN-END pairs, particularly when such a pair creates a new block. As the compiler discovers BEGIN's in a program it numbers them consecutively starting at one and each END is given the same number as the BEGIN which it matches. The appropriate diagnostic is

B n or E n

where n is the number assigned by the compiler. If the outermost BEGIN-END pair is omitted, BO appears at the beginning of the program.

Blocks are also numbered in order of appearance in the program and the extent of a block is denoted by the diagnostics

BLOCK n and END BLOCK n

which occur in matched pairs.

On the same line as BLOCK n, the message

LEVEL m

is given to indicate how blocks are nested. The outermost block is at LEVEL 1 and each new level causes the level number to be increased by one. At the end of a block the level number is decreased by one.

The following table describes special features of Case Algol that generate diagnostics.

Diagnostic	Description
C	comment occurring immediately after an END
D	double precision constant
FR	forward reference - the compiler has taken an identifier to be a label the definition of which has not yet been encountered (not applicable to switch declarations)
GV	generalized variable operation (see Appendix III)
N	identifier or constant extends over card boundaries
OCTAL	octal constant (operative only under K option)
Q	string constant or format mode extending over card boundaries
T	line terminated or page ejected via exclamation sign (!)
TRACE	trace mode initiated or terminated

COMPILER ERROR MESSAGES

Although the generation of diagnostic messages never interferes with the production of an object program, the appearance of error messages during

compilation may do so. These messages are intended to be self-explanatory and will not be described in detail here. However the programmer should be aware of the following conditions:

1) The compiler indicates with an asterisk (*) the place where the existence of an error was definitely ascertained. However, the actual programming error may well have occurred some distance before the point that is marked with an asterisk. The programmer should search backwards from the asterisk for his mistake.
2) When an error in syntax is found, the compiler may stop looking for other errors in the current Algol instruction. Therefore it may take more than one compilation for all syntactical errors to be rooted out.
3) A single error may cause several messages to be printed, some of which may be spurious. The correction of such an error may eliminate several messages from the next compilation.
4) When the A option is in effect, an attempt will be made to produce an object program in spite of syntactical errors, and some approximation to a normal object program will be placed in the user's program complex on drum, even though it may not be executable. If the A option is <u>not</u> in effect and no object program is generated, this fact will be announced by the compiler at the end of the listing.

ERROR MESSAGES AT EXECUTION TIME

When an Algol-compiled program is executed, the system always makes available library routines which can provide information about errors caused by faulty programming.

In the event of an error during a run, control is transferred to the run time error routine. This routine prints the contents of the thin-film registers (B, A, R) if the N option is not selected by the XQT card (see Chapter XIV, THE XQT CARD). The appropriate error message is printed together with the location from which the call was made to the routine in which the error was detected.

If the error occurs in a procedure, the procedure name (to six characters) and the complete nesting of procedures, of which this procedure is the innermost procedure, is printed along with the location from which each of these procedures is called. For example, consider the following program (all sample programs in this chapter are assumed to have been compiled under the name PROG):

EXAMPLE 1:

```
1           REAL A, B, C $
2           REAL PROCEDURE PROC1(X,Y) $
3           VALUE X, Y $ REAL X, Y $
4           BEGIN PROC1 = X/Y END PROC1 $
5           REAL PROCEDURE PROC2(X,Y) $
6           VALUE X, Y $ REAL X, Y $
7           BEGIN PROC2 = X*Y END PROC2 $
8           REAL PROCEDURE PROC3(X,Y) $
9           VALUE X, Y $ REAL X, Y $
10          BEGIN PROC3 = X+Y END PROC3 $
11          REAL PROCEDURE PROC4(X,Y) $
12          VALUE X, Y $ REAL X, Y $
13          BEGIN PROC4 = X-Y END PROC4 $
14          A = 1.0 $ B = 0.0 $
15          C = PROC3(PROC2(A,B),PROC4(PROC1(PROC3(A,B),B),A))$
                                          *
```

Notice that the call to PROC1 shown by the asterisk (*) causes a divide overflow (division by zero). The resulting error message is:

```
INFINITY, INFINITY, YOU DONE DIVIDED BY ZERO
PROG     ON LINE 4
PROC1 DEFINED AT PROG ON LINE 2, CALLED FROM PROG ON LINE 15
PROC4 DEFINED AT PROG ON LINE 11, CALLED FROM PROG ON LINE 15
PROC3 DEFINED AT PROG ON LINE 8, CALLED FROM PROG ON LINE 15
```

This error message tells us that the error occurred in PROC1, which is nested in PROC4, which in turn is nested in PROC3.

Following is a complete list of run time error messages generated by the ALGOL library, along with their respective meanings and causes.

INTERNAL ERROR

This message results from a fault of the library or the compiler. It may be forced, however by improper use of the language. Rewriting the seation of coding in which the error occurred will usually resolve this error.

INCORRECT NUMBER OF ARGUMENTS

This error occurs when the number of arguments supplied as parameters to a procedure or to a library routine is not compatible with the number of arguments required by the procedure or the library routine. For example, consider the following program:

EXAMPLE 2:

```
1       REAL FAUTE $
2       PROCEDURE DUMMY (F) $
3       REAL PROCEDURE F $
4       FAUTE = F(1.0, 2.0) $
5       DUMMY  (COS) $
```

The COS routine requires one argument. Specifying two parameters generates the message:

```
INCORRECT NUMBER OF ARGUMENTS TO COS
PROG ON LINE NO. 2
DUMMY DEFINED AT PROG ON LINE NO. 2, CALLED FROM PROG ON LINE NO. 5
```

The message INCORRECT NUMBER OF ARGUMENTS can also arise from a call to a procedure declared within or external to the main Algol program. In either case the message would read:

```
INCORRECT NUMBER OF ARGUMENTS TO PROCEDURE AT (LOC.)
```

EXAMPLE 3:

```
1       REAL X, Y, Z $
2       REAL PROCEDURE WOOPS (X,Y,Z) $
3       REAL X, Y, Z $
4       WOOPS = X - Y + Z $
5       Z = WOOPS (X,Y) $
```

```
INCORRECT NUMBER OF ARGUMENTS TO PROCEDURE
PROG ON LINE NO. 5
WOOPS DEFINED AT PROG ON LINE NO. 2, CALLED FROM PROG ON LINE NO. 5
```

```
BOOM!!! MEMORY CAPACITY EXCEEDED
```

No, you did not really blow up the computer. However, the storage area available for your program has been exceeded. The message can occur when space is being allotted for single variables, arrays, strings, or blocks.

EXAMPLE 4:

```
REAL ARRAY ROOF(1..260000)
```

The resulting error message is:

```
BOOM!!! MEMORY CAPACITY EXCEEDED IN ARRAY DECLARATION
```

The message for each case of this error is:

BOOM!! MEMORY CAPACITY EXCEEDED IN...	
STRING DECLARATION...	For storage of strings
ARRAY DECLARATION...	For storage of arrays
PROCEDURE...	For Blocks and procedure blocks

IMPROPER ARRAY DECLARATION

This message results from improperly specifying the limits on an array subscript by declaring the lower limit greater than the upper limit. The message gives the array name and the bounds on the improperly specified subscript.

EXAMPLE 5:

```
1       INTEGER M, N $
2       M = 2 $ N = 1 $
3       BEGIN REAL ARRAY NOTAGAIN(M..N) $
4       END
        IMPROPER ARRAY DECLARATION NOTAGA ( 2 : 1 )
        PROG ON LINE NO. 3
```

NOTE: Other forms of the above message are:

```
IMPROPER OWN ARRAY DECLARATION...
IMPROPER STRING ARRAY DECLARATION...
IMPROPER OWN STRING ARRAY DECLARATION...
```

They have the same meaning as IMPROPER ARRAY DECLARATION.

SUBSCRIPT OUT OF RANGE FOR ARRAY

This message results from a subscript specified for a given array not

being within the boundaries declared for this array. The message gives the following information:

1) Which subscript is out of range
2) The value of the subscript out of range
3) The first six characters of the array name if it is not a string
4) The subscript bounds of the subscript out of range
5) The location from which the call was made to the array

For example, consider the following program:

EXAMPLE 6:

```
1           REAL B $
2           REAL ARRAY SOMENAME (1..4,-3..7) $
3           B = SOMENAME (0,-2) $
```

The resulting error message is:

```
SUBSCRIPT NO.  1 (VALUE OF   0) OUT OF RANGE FOR ARRAY SOMENA (1:4)
PROG ON LINE NO. 3
```

Notice that subscript number one, counting from the left, is not within the bounds specified for array SOMENAME, for the subscript has a value of 0, and the bounds specified for this subscript are (1..4).

EXAMPLE 7:

```
1           REAL B $
2           REAL ARRAY DATAHOLD (1..4,3..7) $
3           B = DATAHOLD (3,-17) $
```

The resulting error message is:

```
SUBSCRIPT NO.  2 (VALUE OF   -17) OUT OF RANGE FOR ARRAY DATAHO (3:7)
PROG ON LINE NO. 3
```

SUBSCRIPT OUT OF RANGE FOR STRING ARRAY

This message occurs when (1) a subscript to the "array portion" of a string array is out of range, as explained above for strings and as shown in EXAMPLE 7, or (2) the subscript designating the character position in a string element exceeds the bounds of that string element, as shown in EXAMPLE 8 and EXAMPLE 9.

EXAMPLE 8:

```
1           STRING A (10) $
```

```
2       STRING ARRAY ITHURTS (23..1..10) $
3       A(1,8) = ITHURTS (13,8..12) $

        SUBSCRIPT NO. 1 (VALUE OF 12) OUT OF RANGE FOR STRING ARRAY
        LIMITS WERE (1:10)
        PROG ON LINE 3
```

EXAMPLE 9:

```
1       STRING HELP  (10) $
2       STRING ARRAY ITHURTSMORE (23..1..10) $
3       HELP (1,8) = ITHURTSMORE (25,8..3) $

        SUBSCRIPT OUT OF RANGE FOR STRING ARRAY, VALUE OF (25,8), LENGTH WAS 23
        PROG ON LINE 3
```

EXAMPLE 10:

```
1       STRING ALONG (10) $
2       STRING ARRAY OHNO (23..1..10) $
3       ALONG (1,8) = OHNO (21,8..3) $

        SUBSCRIPT OUT OF RANGE FOR STRING ARRAY, VALUE OF (21,8), LENGTH WAS 23
        PROG ON LINE 3
```

SUBSCRIPT OUT OF RANGE FOR STRING VARIABLE

This message occurs when the subscript specifying the character position in a string variable exceeds the length of that string variable.

EXAMPLE 11:

```
1       STRING IGIVEUP (40) $
2       IGIVEUP (50) = 'HE WHO IS CARELESS WILL BE CAST ASIDE...' $

        SUBSCRIPT OUT OF RANGE FOR STRING VARIABLE, VALUE OF (50), LENGTH WAS 40
        PROG ON LINE 2
```

IMPROPER NUMBER OF DIMENSIONS FOR ARRAY

This error occurs when the number of subscripts specified in a call to an array is not equal to the number of dimensions for which the array is defined. The error message gives the following information:

1) The number of subscripts given
2) The dimensionality of the array

3) The first six characters of the array name

4) The location from which the call was made to the subscript calculator

Consider the following program:

EXAMPLE 12:

```
1       REAL ARRAY WHATSIT (1..10) $
2       REAL B $
3       B = WHATSIT (1,3) $
```

The resulting error message is:

```
IMPROPER NUMBER OF DIMENSIONS FOR 1 DIMENSIONAL ARRAY,
TWO DIMENSIONS GIVEN AS PARAMETERS TO ARRAY WHATSI
PROG ON LINE 3
```

Array WHATSIT is defined over one dimension. Two subscripts are specified in the array call, resulting in the error.

EXAMPLE 13:

```
1       REAL B $
2       REAL ARRAYXFIELD (1..3,1..4,0..7) $
3       B = FIELDX(2,3) $
```

The resulting error message is:

```
IMPROPER NUMBER OF DIMENSIONS FOR 3 DIMENSIONAL ARRAY,
TWO DIMENSIONS GIVEN AS PARAMETERS TO ARRAY FIELDX
PROG ON LINE 3
```

IMPROPER NUMBER OF DIMENSIONS FOR STRING ARRAY

This error occurs when the number of dimensions given as parameters to a call on a given string array is not equal to the dimensionality of this string array. The information given in the error message is:

1) The declared dimensionality of the array

2) The first six characters of the array name

3) The location from which the call was made to the string array subscript calculator which discovered the error.

EXAMPLE 14:

```
1       STRING WHAT (10) $
2       STRING ARRAY IT (20..1..10,1..20) $
3       WHAT (1,2) = IT (5,2..4)
```

IMPROPER NUMBER OF DIMENSIONS FOR 2 DIMENSIONAL ARRAY 1 DIMENSIONS GIVEN AS PARAMETERS
PROG ON LINE 3

NOTE: The error message for OWN STRING ARRAY is exactly the same as the message for STRING ARRAY, if the error is of the type "IMPROPER NUMBER OF DIMENSIONS" or "SUBSCRIPT OUT OF RANGE".

Similarly, the error message for OWN ARRAY is exactly the same as the message for ARRAY, if the error is of the type "IMPROPER NUMBER OF DIMENSIONS" or "SUBSCRIPT OUT OF RANGE".

RESULT UNDEFINED

A message of this type occurs when, for a given argument, a specified library procedure is unable to produce a result. For example consider the following subprogram:

EXAMPLE 15:

```
1       REAL A $
2       A = SQRT (-1.0) $
```

Since for real arithmetic the square root of a negative number is undefined, the following message results:

RESULT UNDEFINED FOR SQRT, ARGUMENT=-0.10000000+01
PROG ON LINE 2

Note: The value of the argument is interpreted as a decimal number times a signed power of 10.

EXAMPLE 16:

```
1       REAL A $
2       A = ARCSIN(2.0) $
```

The resulting error message is:

RESULT UNDEFINED FOR ARCSIN, ARGUMENT= 0.20000000+01
PROG ON LINE 2

The message RESULT UNDEFINED can also occur when, in attempting to convert a string to an integer, the string contains characters which are not numerics. For example, consider the following subprogram:

EXAMPLE 17:

```
1       INTEGER A $
2       STRING S(60) $
3       S(1) = 'IF ALL ELSE FAILS...' $
4       A = INTEGER (S(1,8)) $
```

RESULT UNDEFINED FOR STRING TO INTEGER

PROG ON LINE 4

The message RESULT UNDEFINED can also originate in one of the power routines. No result is defined for zero to the zero power. As a typical example, consider the following subprogram:

EXAMPLE 18:

```
1       REAL A $
2       A = 0.0**0.0 $
```

Both the base and the exponent are of type real. Hence, the error message would read:

RESULT UNDEFINED FOR REAL TO REAL POWER, BASE = 0.00000000+00,
EXPONENT = 0.00000000+00
PROG ON LINE 2

Note: When an integer is raised to a real power, the integer base is converted to type real. The real to real power routine is then called. If the result is undefined for an integer to a real power, therefore, the error message would read:

RESULT UNDEFINED FOR REAL TO REAL POWER...

Although zero to a negative power gives a result without bounds, we give the message "RESULT UNDEFINED".

EXAMPLE 19:

```
1       REAL A $
2       A = 0.0**-3.0 $
```

RESULT UNDEFINED FOR REAL TO REAL POWER, BASE= 0.00000000+00,
EXPONENT=-0.30000000+01
PROG ON LINE 2

RESULT OUT OF RANGE

This message occurs when the resulting absolute value of a given library routine upon a given argument is greater than 10.0^{38} or $2^{35}-1$, for results of type REAL and of type INTEGER respectively.

EXAMPLE 20:

```
1       REAL A $
2       A = EXP(90.0) $
```

RESULT OUT OF RANGE FOR EXP, ARGUMENT=0.90000000+02
PROG ON LINE 2

Note: The exponential function is within the range of the computer when its argument is less than 88.028.

EXAMPLE 21:

```
1       INTEGER A $
2       A = ENTIER(1.2&12) $
    RESULT OUT OF RANGE FOR ENTIER, ARGUMENT =0.12000000+13
    PROG ON LINE 2
```

The message RESULT OUT OF RANGE can also occur when, in converting a string to an integer, the result is greater than $2^{35}-1$.

EXAMPLE 22:

```
1       INTEGER A $
2       A = INTEGER('123456789123') $
    RESULT OUT OF RANGE FOR STRING TO INTEGER
    PROG ON LINE 2
```

The message RESULT OUT OF RANGE can occur in one of the power routines. In this case, it means that the result of raising a given base to a given power is not within the numeric range of the computer, as outlined above. For a typical example, consider the following:

EXAMPLE 23:

```
1       INTEGER A $
2       A = 5**37 $
    RESULT OUT OF RANGE FOR INTEGER TO INTEGER POWER, BASE=        5,
    EXPONENT=            37
    PROG ON LINE 2
```

Note: A real number raised to a real power, resulting in an absolute value of less than 10.0^{-38}, yields a result of 0.0. An integer raised to an integer power resulting in an absolute value of less than 1 yields result of 0.

The real to integer power routine may give an error message of RESULT OUT OF RANGE if the result is less than 10.0^{-38} in absolute value.

In raising an integer to a real power, the integer is converted to type real. The real to real power routine is then called. Hence, a message of RESULT OUT OF RANGE FOR REAL TO REAL POWER can originate in raising an integer to a real power, the result of which is out of range as previously defined.

INFINITY, INFINITY, YOU DONE DIVIDED BY ZERO

This message originates when a division by zero is attempted. Do not forget that a real number less than 10.0^{-38} in absolute value will assume the value of zero.

HURTSVILLE, CHARACTERISTIC OVERFLOW

This message occurs when the result of an arithmetic operation (+,-,*,/) is of the type REAL or REAL2 and is greater than 10.0^{38} in absolute value. The message can also result from raising a real number to an integer power. If the power is less than 64 and the result of the exponentiation is out of range, the message HURTSVILLE, CHARACTERISTIC OVERFLOW will result.

To avoid the error HURTSVILLE, CHARACTERISTIC OVERFLOW, it is sometimes possible to "break up" an arithmetic expression such that at no time during the evaluation of this expression will the absolute value of any part of it exceed 10.0^{38}. If the aforementioned scheme does not work, an insertion of a dummy variable may be the key to your problem.

EXAMPLE 24:

```
1       REAL A, B, C, D $
2       A = 1.0&-25 $
3       B = 1.0&30 $
4       C = 1.0&21 $
5       D = (B/A)/C $
    HURTSVILLE, CHARACTERISTIC OVERFLOW   FOR    0.10000000+31    /    0.09999999-
    PROG ON LINE 5
```

Solution to EXAMPLE 24:

EXAMPLE 25:

```
1       REAL A, B, C, D $
2       A = 1.0&-25 $
3       B = 1.0&30 $
4       C = 1.0&21 $
5       D = B/(A*C)
```

UNRECOVERABLE TAPE/DRUM ERROR

This message occurs when, during a tape or drum operation, an unsuccessful

read or write occurs. The error could be the result of:

1) A hardware failure.
2) The user's attempt to read information which is not present where he is looking for it.

ATTEMPT TO PASS END OF INFORMATION IN READ

This error occurs when an attempt is made to read beyond the end of recorded information on magnetic tape.

MISSING OR MISPLACED ACTIVATION PHRASE

This message may result from a card read in which the supplied format does not have a proper activation phrase (see Chapter IX, FORMAT PHRASES).

EXAMPLE 26:

```
1   INTEGER A $
2   FORMAT FORM(I8,A) $
3   READ(FORM,A) $
    MISSING OR MISPLACED ACTIVATION PHRASE IN READ
    PROG ON LINE NO. 3
    READ, CALLED FROM PROG ON LINE NO. 3
```

CONSTANT OUT OF RANGE

This error occurs when:

1) In a read, an integer constant is encountered which is greater than $2^{35}-1$ in absolute value.
2) In a read, a real constant is encountered which is greater than 10.0^{38} in absolute value.
3) A string exceeds 4095 characters in length.

ILLEGAL CHARACTER/UNDEFINED TYPE CONVERSION IN ABOVE RECORD

In attempting a read, one of the following occurred:

1) An illegal character was encountered within a record. For example, an attempt to read the word 1ØH1 as a real number would result in the above error.

2) A word is encountered which is not of the type requested and for which a conversion to the requested type is not defined. For example, consider the following:

EXAMPLE 27:

```
1        BOOLEAN NEVERAGAIN $
2        READ (NEVERAGAIN) $
```

Encountering an integer other than Ø or 1 would cause the above mentioned error, for no conversion is defined from integer (other than Ø or 1) to Boolean.

NOTE: Above the register printout (if present) will be found a listing of the data card which caused the error. The asterisk (*) points to the character on the card which ultimately caused transfer to be made to the error routine.

IMPROPER PARAMETER

This error occurs when the parameter supplied to a procedure or to a library routine is not of the type required.

Case 1: Call by name parameter to procedure.

EXAMPLE 28:

```
1   REAL PRUNEJUICE, APPLEJUICE, LEMONJUICE $
2   PROCEDURE JUICY(BOT, TOMS, UP) $
3   REAL BOT, TOMS $
4   INTEGER UP $
5   BEGIN UP = ENTIER(TOMS+BOT)
6   END JUICY $
7   JUICY (PRUNEJUICE, APPLEJUICE, LEMONJUICE) $
```

Since the call by name parameter, LEMONJUICE, is not of the same arithmetic as the variable UP, the error "IMPROPER PARAMETER TO PROCEDURE" will result.

Case 2: A library routine is used as a call by name parameter and the argument supplied to this library procedure is of the improper type.

EXAMPLE 29:

```
1       PROCEDURE F (Q) $
2       REAL PROCEDURE Q $
3       BEGIN REAL A $
4       A = Q('ERROR') END $
5       F(SIN) $
```

IMPROPER PARAMETER TO SIN
PROG ON LINE NO. 4
F DEFINED AT PROG ON LINE NO. 1, CALLED FROM PROG ON LINE NO. 5

NOTE: An improper parameter to tape and drum routines results in the message "IMPROPER PARAMETER TO (NAME LOST AT OCCURRENCE OF ERROR) AT (LOC)

Case 3. The form of the READ statement is incorrect.

EXAMPLE 30:

```
1       STRING A(1) $
2       A = 'S' $
3       READ (CARDS,TAPE(A)) $
```

Two input devices are specified in the READ statement. The error message would read:

IMPROPER PARAMETER TO READ
PROG ON LINE NO. 3
READ, CALLED FROM PROG ON LINE NO. 3

UNDEFINED TYPE CONVERSION

In attempting to convert a variable of a given type into a different type, it is discovered that no conversion is defined between the two types. For example, consider the following:

EXAMPLE 31:

```
1       STRING A(2) $
2       FORMAT GREEN (R10.2,A1) $
3       LIST BEAN(A) $
4       A = 'SS' $
5       WRITE (GREEN,BEAN) $
```

No conversion is defined from STRING to REAL. Hence the error:

UNDEFINED TYPE CONVERSION IN WRITE
PROG ON LINE NO. 5
WRITE, CALLED FROM PROG ON LINE NO. 5

There are a number of error messages which can result from an error in the use of the FORMAT routine. These messages have the same meaning as the corresponding messages for compilation errors. Furthermore, they contain the additional information which is always provided by the error routine.

NOTE: The abbreviations "D.P." and "REAL2" both characterize a double precision routine. For example, the message "RESULT UNDEFINED FOR D.P. TO D.P. POWER..." means that the error occurred in the routine for raising a double precision base to a double precision power.

INSUFFICIENT DATA

If a READ statement does not find enough data on the (card reader) input device to satisfy the input parameter list, then the execution will be terminated with the following message:

```
INSUFFICIENT DATA FOR PROGRAM
PROGRAM ABNORMALLY ABANDONED
```

ERROR NUMBERS FOR LIBRARY ERROR MESSAGES

There is associated with each type of message an error number that can be used in conjunction with the ERROR and ERRORTRAP procedures (q.v.). The following is a list of error message types and their associated error numbers that are currently part of the Algol library.

ERROR NUMBER	DESCRIPTION
0	Internal error
1	Incorrect number of arguments
2	Memory capacity exceeded
3	Undefined designational expression
4	Improper size phrase
5	Undefined type conversion
6	Same as 4
7	Same as 4
8	Not used
9	Unrecoverable Tape/Drum error
10	Attempt to pass end-of-information
11	Constant out of range

ERROR NUMBER	DESCRIPTION
12	Not used
13	Characteristic overflow
14	Divide by zero
15	Improper number of dimensions
16	Not used
17	Result undefined
18	Result out of range
19	Real2/Complex/Misc routine not in library
20	Illegal character/Undefined type conversion
21	Improper sequence of format phrases
22	Improper parameter
23	String too long
24	Extra right parenthesis
25	Missing or misplaced activation phrase
26	Improper format symbol
27	Extra left parenthesis
28	Subscript out of range
29	Subscript out of range for string array
30	Not used
31	NO MESSAGE IS PRINTED

DIAGNOSTIC PROCEDURES

Two special procedures, DUMP and ERRORTRAP, are available in the Algol library to speed up the process of debugging a program. DUMP allows the user to gather information about the status of a program when an error occurs at execution time. ERRORTRAP allows the user to replace the usual library error routines with an Algol procedure of his own. With this feature the programmer can gather special information and decide how execution should continue in the event of an error.

THE DUMP STATEMENT

The programmer may specify that the values of certain variables be printed out when a run-time error occurs by using the DUMP statement.

GENERAL FORM:

DUMP(<dump list>)

where: <dump list> is <identifier> or
<dump list>,<identifier>.

Each <identifier> is the name of a simple variable or the name of an array, and further, <identifier> must not be a call by name parameter. Note that constants, expressions, subscripted arrays, or subscripted strings may not be used in a DUMP statement.

Any block can have only one dump list invoked at a given time. Each DUMP statement (in effect) erases the old <dump list> for that block and replaces it with a new list.

Each active block can have a <dump list> associated with it. If an error occurs while N blocks are active, then a maximum of N<dump list>'s will be printed out. If no error occurs after the DUMP statement is encountered, the <identifier list> will not be printed out.

EXAMPLE:

```
INTEGER A,B,C,D,E,F,H,J,K,X$
DUMP(A,B,C,D,E,F,H,J,K,X)$
A = 1 $ B = 2 $ C = 3 $ D = 4 $ E = 5 $
F = 6 $ H = 7 $ J = 8 $ K = 9 $ X = 0 $
H = A*B/X
```

When execution of this program is attempted, the results are as follows:

```
INFINITY, INFINITY, YOU DONE DIVIDED BY ZERO
PROG ON LINE NO. 5


DUMP LIST:

A       1
B       2
C       3
D       4
E       5
F       6
H       7
J       8
K       9
X       0
```

THE ERRORTRAP PROCEDURE

When a run-time error occurs the Algol library gathers and prints as much information as it can about the situation that caused the error and then terminates the execution.

By using ERRORTRAP the programmer may cause control to be transferred to a procedure of his own before the library routine takes over. His procedure may then decide on the basis of the error type how to continue.

GENERAL FORM OF THE CALL

ERRORTRAP (<proc name>)

where <proc name> is the identifier of the user-written procedure to which control will be transferred if an error occurs. This procedure must have exactly one parameter, of type INTEGER. On entry to the user-written procedure, the value of the integer parameter will be set equal to the appropriate error number by the ERRORTRAP routine (see the accompanying table of library error messages). Error number 2, Memory Capacity Exceeded, cannot be handled this way.

The ERRORTRAP routine may be disabled by calling it with no parameter:

ERRORTRAP

When this call is in effect, an error at execution time will be handled in the usual way, i.e., the library error routine will take control and the user's error-handling procedure will be ignored.

If the user-written procedure is exited via the procedure END rather than by a GOTO statement, then control is passed to the library error routine and normal error processing ensues.

Unless the programmer wishes to live dangerously, the first statement of his error-handling procedure will be:

ERRORTRAP

to disable his errortrap procedure. If errortrap is not disabled and an error is detected in the error-handling procedure, then the error-handling routine will be called again and possibly the error will be detected again, and round and round... Thus the error routine may be entered recursively with (possibly) the wrong value as parameter.

THE ERROR PROCEDURE

The ERROR procedure allows an Algol program to call in the library error routine even though the system library itself has not detected an error. This can be particularly useful if the Algol program is intended to be a processor.

GENERAL FORM OF THE CALL:

ERROR (<arith exp>)

The effect of this call is to transfer control to the library error routine with the value of <arith exp> as the error number. <arith exp> is converted to type INTEGER if it is not already of that type. The library error routine takes action appropriate to the value of the parameter, including printing messages, unwinding nested procedures, and terminating the execution.

TRACE OPTIONS

The compiler allows the programmer to exercise options that affect the compilation and execution of an Algol program. The following options are currently available:

ALGOL COMPILATION OPTIONS AND TRACE NUMBERS

OPTION LETTER	TRACE NUMBER	DESCRIPTION
A	-	Accept program even if errors are found (an element is placed in program complex
B	21	Block and BEGIN-END diagnostics suppress
C	-	Cycle numbers suppressed
D	29	Do not abort compilation after 20 errors
E	-	Externally define all procedures declared in block one
F	-	Full card (80 columns) scanned
H	-	Hierarchy of operators changed so that multiplication is performed before division (on the same parenthesis level)
J	-	Treat program as processor (contents of B11 saved)
K	25	Octal constants allowed
L	1	List edited machine language

OPTION LETTER	TRACE NUMBER	DESCRIPTION
N	13	Suppress listing (except errors)
O	16	Open coding - array subscripts not checked
R	18	Registers dumped in case of errors at compile time
S	-	Card number tables treated as labeled common block
V	26	Number errors in order of occurrence
W	-	Print correction cards before source listing
X	12	Abort compilation immediately if error is detected
Z	15	Do not generate card number tables
-	28	Externally define all OWN variables
-	30	Allow G.V. array operations

Trace option letters are punched on the Algol processor card starting in column two (see EXEC III Manual, p. 50-54). This causes the option to be in effect throughout the compilation process.

Some of the options may be invoked selectively during compilation by means of the corresponding trace number. The following Algol statement will turn on the designated trace options:

TRACE ON $n_1, n_2, \ldots, n_K$ $

where n_I are unsigned integers chosen from the above table. The following statement turns off the designated options:

TRACE OFF $n_1, n_2, \ldots, n_K$ $

To turn off all trace options that are not being used as letter options in this compilation, the statement

TRACE OFF $

is sufficient.

The case $n_I = 0$ does not have a corresponding option letter. The statements

TRACE ON 0 $ and TRACE OFF 0 $

will cause the compiler to insert the following 1107 Sleuth statements, respectively, into the object program:

SLJ XLOG$ and SLJ XFREE$

(see 1107 Monitor System Notes for an explanation of these diagnostic traces).

The statements

TRACE ON $n_1,...n_K$ $

TRACE OFF $n_1,...n_K$ $

may appear in a program anywhere a COMMENT may appear.

XIV...

USING ALGOL UNDER EXEC III

The programming of the Case 1107 makes use of various elements of a comprehensive operating system. The operating system includes the Algol compiler and library, and is controlled by a basic set of routines known as Exec III.

In particular, compilation and execution of an Algol program are governed by the use of Exec control cards. A full description of control cards that are recognized by the executive routines is given in the Exec III manual. The present discussion is limited to some of the basic features used in Algol programming.

EXEC CONTROL CARDS

Every Exec control card must have a 7/8 punch in column one and any card with a 7/8 punch in column 1 will be treated as an Exec control card. This multiple punch is often referred to as a "master space". In the following discussion the symbol "␣" represents a blank card column.

The first card of every deck that is read into the computer should be either a RUN card, LST card or PCH card. The LST and PCH cards will be explained later. If compilation and/or execution of a program are desired, the RUN card must be used.

THE RUN CARD

A typical RUN card for a student run is as follows:

$\frac{7}{8}$ △ RUN △ 12345, E12

where the student number is 12345 and the class for which the program is

being run is E12. Each class using the computer is assigned time and page limits by its instructor.

A typical RUN card for a project run is:

7/8 Δ RUN Δ 99999,2

where the project number is 99999 and the programmer number is 2. If, as in the above example, no time or page limit is declared then the assumed limits for that project and programmer will be in force. To specify limits other than the assumed limits, for example, a maximum of 120 seconds and 20 pages, the card would appear as follows:

7/8 Δ RUN Δ 99999,2, (120,20)

THE ALG CARD

To initiate compilation of an Algol program, the source deck must be immediately preceded by an Algol processor card. This is an Exec control card that specifies the name of the program and any Algol options the programmer wants to exercise. An example of an Algol processor card is:

7/8FN Δ ALG Δ PROG1

where the program is named PROG1 and the F and N options are indicated.

If several programs are to be compiled in the same run, a separate Algol processor card must precede each program.

THE XQT CARD

If, after compilation, the program is to be executed, the Algol source deck is followed by an "execute" card. This Exec control card specifies the name of the program to be run and usually the N option is exercised on this card. If the program name is PROG1 the card is:

7/8N Δ XQT Δ PROG1

If the N option is left off the XQT card the allocation of computer memory for the main program and its associated procedures will be printed out, thereby providing one or more pages of information that is almost always worthless.

DATA CARDS AND THE EOF CARD

The XQT card is followed by the data cards used by the program. Among the data cards may be EOF cards, to be used in conjunction with the input routine described in Chapter IX. An EOF card has the form:

7/8 Δ EOF Δ

Columns two and six of an EOF card must be blank but any 1107 characters may be punched in columns 7-80

THE FIN CARD

The last card of an input deck must be a FIN card. This Exec control card signals the end of the user's file and the entire input deck will be ignored if it is not present. Its form is:

7/8 Δ FIN Δ

Just as with the EOF card, columns two and six of a FIN card must be blank and columns 7-80 may contain any 1107 characters.

If a program is to be executed the FIN card follows the data cards or, if there are none, it follows the XQT card. A program may be compiled without being executed, in which case the FIN card may immediately follow the source deck.

SAMPLE INPUT DECK

7/8 Δ RUN Δ 12345, E12

7/8F Δ ALG Δ TEST1

```
Δ RUN 76344,E12
ΔF ALG TEST1

COMMENT  THE F OPTION ON THE ALG CARD TELLS THE COMPILER TO
         SCAN ALL 80 COLUMNS OF THE SOURCE CARD, NOT JUST
         THE FIRST 72 $
INTEGER I,START,QUIT,THISMUCH$
REAL PROCEDURE FACTORIAL2(N)$
   VALUE N$  INTEGER N$
   FACTORIAL2 = IF N EQL 0 THEN 1.0
                          ELSE N*FACTORIAL2(N-1)$
READ(START,THISMUCH,QUIT)$
FOR I = START STEP THISMUCH UNTIL QUIT DO
                                  WRITE(FACTORIAL2(I))$

ΔN XQT TEST1
   2    3    10
Δ FIN
```

When the above program is executed, the printout will contain the values of 2!, 5!, and 8!.

THE LST AND PCH CARDS

A deck of cards may be listed on a printer using the LST card. The form of this card is:

7/8 ΔLSTΔ

Any deck of cards <u>not</u> including a RUN or FIN card may be listed by putting a LST card in front of it and a FIN card behind the deck. When this deck is read into the computer the listing will appear on the printer corresponding to that reader.

The PCH card is used to duplicate a deck of cards. The form of this card is:

7/8 △PCH△

The use of the PCH card is similar to that of the LST card except that the output consists of punched cards instead of printout. Note that the punch feed should be loaded with enough cards to accommodate duplication of the entire input deck.

EXAMPLE:

7/8 △LST△

deck to be listed

7/8 △FIN△

THE COMPLEX UTILITY ROUTINE

The 1107 operating system includes a processor called the Complex Utility Routine (CUR). It may be used to store a library of source and/or object programs on magnetic tape, retrieve them from tape when needed, and punch object programs onto cards.

CUR and other parts of the operating system are described in the manual, 1107 Monitor System Notes.

Appendix I

SPECIAL IDENTIFIERS

Two classes of identifiers, reserved words and predefined identifiers, are treated as special cases by the Algol compiler. These identifiers are underlined in this manual to call the reader's attention to their use.

RESERVED WORDS

A reserved word is an identifier whose use in Algol programming is restricted to the situations described in this manual. The compiler will not allow the meaning of a reserved word to be changed by the programmer (e.g., a reserved word cannot be used as variable name, procedure name or label).

The following is a list of reserved words:

AND	LABEL
ARRAY	LEQ
BEGIN	LIST
BOOLEAN	LISTSTRUCTURE
COMMENT	LOCAL
COMPLEX	LSS
DEFINE	NEQ
DO	NOT
DUMP	OR
ELSE	OWN
END	PROCEDURE
EQIV	RANK
EQL	REAL
EQUIV	REAL2
EXTERNAL	STEP
FALSE	STRING
FOR	SWITCH
FORMAT	THEN
GEQ	TO
GO	TRACE
GOTO	TRUE

GTR	UNTIL
IF	VALUE
IMPL	WHILE
INTEGER	XOR

PREDEFINED IDENTIFIERS

A predefined identifier is an identifier for which a standard definition is known by the compiler (e.g., SIN, READ, etc.). If the programmer wants to use the standard definition of such a word, he must not attempt to redefine its meaning. However, he may redefine the meaning of a predefined identifier in any block in which he will not use it in the standard sense. These identifiers are considered to be defined in block 0.

The following is a list of predefined identifiers:

INTRINSIC FUNCTIONS

ABS	PARITYODD	DMPVAR
CLOCK	PARTBL	ERROR
CLOK	SETRANK	ERRORTRAP
DIMENSIONS	SIGN	FORCETYPE
DSA	SSA	HEADING
DSC	SSC	INSERT
DSL	SSL	LOWERBOUND
EVEN		MARGIN
EXIT	LIBRARY FUNCTIONS	MERGE
EXITABORT	ALPHABETIC	NOLIST
EXITERROR	CHAIN	NUMERIC
EXITNORMAL	CLOSE	OPTION
FIELD	COREMAX	PAPER
FIELDS	CORETOTAL	PCFELT
IMAG	CRTSW	POSITION
LENGTH	DATE	READ
MCR	DELETE	RELEASE
MEMORY	DEVICE	REWIND
MOD	DISPLAY	QUEUE
ODD	DMPBLK	SORT
PARITYEVEN	DMPCOR	SWAP

TRANSFER
TYPE
UPPERBOUND
WRITE

INPUT-OUTPUT DEVICES

CARDS
CORE
DRUM
EDIT
PCF
PRINTER
PUNCH
SLIP
TAPE

PLOTTER ROUTINES

CHANGEKEY
CHAR
DPARAM
KEYCOUNT
MOVE
NEWGRAPH
PLOT
SCALE
SCHAR
STOP

MATHEMATICAL FUNCTIONS

ARCOS
ARCSIN
ARCTAN
ARG
COS
COSH
ENTIER
EXP
INTRANDOM
LN
MAX
MIN
RANDOM
SIN
SINH
SQRT
TAN
TANH

INPUT-OUTPUT MODIFIERS

EOF
EOI
INTERLOCK
KEY

The following identifiers may be used as the programmer sees fit, but each has a special meaning in the specified context:

FORTRAN or SLEUTH following the word EXTERNAL (used in declaration of external procedure);
ON or OFF following the word TRACE (used to invoke or revoke trace option numbers).

APPENDIX II

WRITING ALGOL PROCEDURES IN 1107 SLEUTH

For the benefit of programmers writing procedures in Sleuth II, this appendix describes the calling sequence generated by the Algol compiler, and explains how the parameters to the procedure may be handled in the Sleuth coding.

THE EXTERNAL SLEUTH PROCEDURE

The calling sequence generated for a call on an EXTERNAL SLEUTH PROCEDURE is:

```
                LMJ         B11, <procedure name>
                +K
                G           T1,A1,L1,ADDRESS1
                .               .
                .               .
                .               .
                G           TK,AK,LK,ADDRESSK
G               FORM        6,3,3,24
```

where K is the number of parameters to the procedure and the word

```
                G           TI,AI,LI,ADDRESSI
```

is the descriptor of the Ith parameter. $ADDRESS_I$ may be any of the following:

```
B10,    ADDRESS RELATIVE TO B10
B9,     ADDRESS RELATIVE TO B9
0,      ABSOLUTE ADDRESS
```

Of the twenty-two bits in $ADDRESS_I$, the high-order four denote the index register and the low-order eighteen bits denote the relative address, h and i bits.

T_I is an octal number that denotes the structure of the Ith parameter:

T_I	Meaning
0	Simple variable or expression
010	ARRAY
011	STRING ARRAY

021	Generative LIST
040	Sort/Merge
041	RANK
042	FORMAT
043	LIST STRUCTURE

Generalized variables are denoted by 040-047, the first four of which are currently predefined via system library routines.

A_I denotes the type of the parameter:

A_I	Meaning
1	INTEGER
2	REAL
3	COMPLEX
4	BOOLEAN
5	STRING
6	REAL2

L_I denotes the nature of the parameter:

L_I	Meaning
0	Simple name
1	Constant
2	Result
3	Not used
4	Localized global name
5	U-field integer constant ($-1 < K$ NO. $< 2^{13}-2$)
6	Indirect result

THE EXTERNAL FORTRAN PROCEDURE

The calling sequence generated for a call on an EXTERNAL FORTRAN PROCEDURE has the same form as the sequence for an EXTERNAL SLEUTH PROCEDURE except that the word specifying the number of parameters

+K

is omitted.

CODING THE EXTERNAL SLEUTH PROCEDURE

The parameters to a Sleuth-written procedure can be made available to the Sleuth routine as follows: The value of the Nth parameter is loaded into a register by the instruction

```
L                <register>,*N,B11
```

If the parameter is a STRING, ARRAY or LIST then the register will be loaded with the descriptor of the parameter (see USE OF STRUCTURED VARIABLES below).

The following suggestions should be observed when coding the Sleuth routine:

1) The entry line must be labeled with the name of the procedure (an identifier of at most six characters) followed by a star (*).
2) Register B10 must not be changed (or chaos will follow).
3) All error checking is the responsibility of the Sleuth routine.
4) If the procedure is being used in the functional sense (e.g., EXTERNAL SLEUTH INTEGER PROCEDURE), the result should be left on exit from the procedure in register A2 (A2, A3 in the case of REAL2 or COMPLEX). If the result is a STRING, ARRAY or LIST the descriptor of the result should be left in A2.
5) The return to the calling program is made by the instruction

   ```
   J             K+1,B11
   ```

   where K is the number of parameters to the procedure.
6) Under all circumstances the contents of registers B1,B2,B3, and B4 should be the same when leaving the Sleuth procedure as when entry was made.
7) Register B9 must not be changed until all the parameters have been accessed.

USE OF STRUCTURED VARIABLES

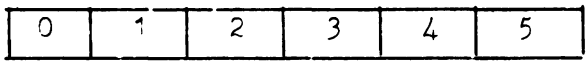
1) Strings

A string is stored in a sequence of memory cells, six characters per cell. The parameter passed to the Sleuth procedure is a string descriptor, which has the form

```
              F           <length>,<position>,<address>
F             FORM        12,6,18
```

where <length> is the number of characters in the string, <position> is

the position of the leftmost character in the first cell, and <address> is the location of the first cell. The sixths of a word are numbered from left to right, from zero to five:



i.e., $0 \leq$ <position> ≤ 5.

2) Arrays

If an array element is used as a parameter to an EXTERNAL SLEUTH PROCEDURE, the value of the element is passed to the procedure. If, however, an array name is used as a parameter, the procedure will receive the address of the first word of the array head. The structure of the array head for non-string arrays is as follows:

<no. of elements>, <address of first element>
<no. words per element>, <no. of dimensions>
<lower (1)>,<upper (1) -lower (1) + 1>
⋮ ⋮
<lower (N)>,<upper (N) -lower (N) + 1>
} Bounds on Subscript Values
<special constant for 0 option>,<address of array name>
<first element of array>
⋮
<last element of array>

The array is stored by rows. For string arrays the structure is:

<no. of elements>,<address of first elt. descriptor>
1, <no. of dimensions>
<lower (1)>,<upper (1) -lower (1) + 1>
⋮
<lower (N),<upper (N) -lower (N) + 1>
+ 0
<descriptor of first element>
<words/string>
<first string>
⋮
<last string>

APPENDIX III

GENERALIZED VARIABLES AND THE DEFINE DECLARATION

A generalized classification scheme for variables has been incorporated in the syntax of Case Algol. The scheme described here has been designed to permit the user to add new classes of variables to his program provided he defines the structure by means of external Sleuth II coding. The structure of these variables is determined by specifying in the Sleuth routines the meaning of the standard operators.

In standard Algol 60, only simple quantities are allowed for, namely, INTEGER, REAL, and BOOLEAN. Variables of type STRING, REAL2, and COMPLEX are found in Case Algol, but these are also handled like simple variables. Arrays are intended only to store large quantities of simple variables, rather than being considered variable entities in themselves.

The introduction of generalized variables is intended to supplement this by allowing the direct manipulation of variables with a non-simple structure, or with attributes that are not common enough to be directly incorporated into the compiler. For example, the manipulation of matrices, polynomials, sets, or topologies could be treated by means of ordinary Algol statements if the proper Sleuth routines were provided.

DECLARATIONS IN CASE ALGOL

Every variable that is used in an Algol program must be declared (i.e. defined) in the following manner:

<classification> <variable$_1$>,...,<variable$_N$>

where <classification> is a sequence of identifiers that names the type and kind of variable and the <variable$_i$>'s denote the variables of that type and kind. The form of <variable$_i$> is:

<name>

or <name>(<parameter list>)

where <name> is the identifier of the variable and <parameter list> consists of a sequence of parameters that may be separated either by commas (',') or colons (':' or '..') Both kinds of separators may be used as, for example, in the standard ARRAY declaration.

THE DEFINE DECLARATION

An identifier that serves as the <classification> of a generalized variable type must be named in a DEFINE declaration to inform the compiler that it is now desirable to have a new kind of variable, viz., <classification> The general form is:

DEFINE <class$_1$>,<class$_2$>,...,<class$_n$> $

where <class$_i$> is:

<classification$_i$>

or <classification$_i$> (N)

where N is an integer constant such that $\emptyset \leq N \leq 7$. Any further occurrence of <classification$_i$> will be treated just as an occurrence of INTEGER, REAL, ARRAY, FORMAT, etc. That is, it serves to identify variables to be declared as being of type and kind <classification$_i$>. It also serves as a transfer function that takes one parameter and converts it to the type and kind of <classification$_i$> (just as INTEGER (1.0) is a call on the transfer function for type and kind INTEGER).

The integer N is called the ranking index. The numerical kind of a generalized variable is 040_8 + N. Systems so far implemented are:

level	kind	Generalized variable
0	040	Sort/Merge, SOL-B
1	041	Ranks
2	042	Formats
3	043	List Structures

Any attempt to define and use generalized variables at these levels while using one of the corresponding predefined generalized variables may result in a certain amount of chaos.

CODING FOR GENERALIZED VARIABLES

Thus, in theory, Algol has now been equipped with the machinery to handle any type of variable. However we must remember that Algol on the 1107 is not purely a meta-language in which we can express an algorithm for the solution of some problem. It also doubles as a source language which is translated into an object code (machine language). Thus we will now

consider what all of the above is translated into.

THE DECLARATION OF A GENERALIZED VARIABLE

In the declaration of a generalized variable the following coding is turned out:

line 1	LX,U	B9,<name>
line 2	LMJ	B11,09---$
line 3	±	n
lines 4 to n+3	<descriptor$_1$> ⋮ <descriptor$_n$>	
line n+4	'<name of the generalized variable>'	

Line 1 loads B9 with the address of the cell associated with the variable. This cell later becomes a pointer to the actual variable. Line 2 transfers control to the Sleuth routine that initializes the generalized variable. The name of the routine is defined by the following algorithm: Take the first three characters from <classification$_1$>. Prefix these characters with 09 and suffix with a $. If there are less than three characters use as many as there are. It is obvious from the above that if two <classification>'s have the same first three characters, there is going to be some confusion. On line 3 n is the number of parameters in this declaration, -n if this variable is OWN, +n otherwise. Lines 4 to n+3 are the descriptors for the n parameters (see Appendix II for the form of descriptors for such a call). The descriptor is in a form such that an indirect address on the word will obtain the parameter. The initial contents of <name> is undefined.

The above calling sequence enables the user to initialize each generalized variable. Initialization consists of three major sections:

1. Setup of structure involved with the manipulation of the variable.
2. It gives the user the location associated with the variable (given in B9). This location is sometimes referred to as the pointer to the variable.
3. It gives the user the opportunity to put the address of an operation table (described below) in the H1 field of the location associated with the variable. The H2 field of this location may be used in any way the user desires. It usually points to

the actual memory locations that the generalized variable occupies, i.e., the address of the variable is put into the H2 field of the pointer.

Consider the following segment of an Algol program (this example is continued later in this appendix):

```
DEFINE  SET(3), ELEMENT, CLASS(4) $
```

(The compiler generates no coding for the above.)

```
ELEMENT A,B$
   LX,U              B9,A
   LMJ               B11,09ELE$
   +                 0
   'A  '
   LX,U              B9,B
   LMJ               B11,09ELE$
   +                 0
   'B  '
```

The allowable operations are:

Binary operators:

+	'ADD'	XOR	'XOR'
-	'SUB'	IMPL	'IMP'
*	'MUL'	EQUIV	'EQV'
/	'DIV'	EQL	'EQL'
//	'DII'	LSS	'LSS'
**	'EXP'	GTR	'GTR'
OR	'OR'	LEQ	'LEQ'
AND	'AND'	NEQ	'NEQ'
=	'REP'	GEQ	'GEQ'

Unary operators:

-	'NEG'	NOT	'NOT'

Subscript calculate: 'SSC'

The address of the operation table for that particular generalized variable should be stored in the H1 field of the pointer.

When an operation occurring in the operation table is requested by an Algol program, control is transferred to the address of the specified routine with the following information available (excluding SSC):

B11 - Address of the first of two (one, if it is a unary operation)

consecutive parameter descriptors; the first descriptor for the left operand and the second for the right operand.

08XIT$ - A location that contains the address to which control should be transferred when done with the operation.

A2,A3 - should contain the result of the operation upon exit. The result is usually a pointer to the resultant quantity if the result is non-simple. If the result is simple, then the appropriate INTEGER, REAL, COMPLEX, BOOLEAN, STRING, or REAL2 quantity should be put in A2,A3. For binary arithmetic operations, the kind of the result will be taken to be the kind of the operand with the larger numerical kind. For Boolean and relational operators, the result kind is always BOOLEAN.

If the operation is 'SSC', the following information is available

B11 - Address of a word which contains an integer which indicates the number of parameter descriptors immediately following.

08XIT$ - contains the return point (see above).

B9 - should contain upon exit the result of the subscripting operation. This result is usually the pointer of the generalized variable referenced by the subscripting operation.

The following conventions must be remembered in the use of subscripted generalized variables:

1) Any variable or expression has type and kind associated with it. A variable is assumed to have universal type if no other type is specified, except for arrays where REAL type is assumed.
2) A variable with a subscript on it drops its kind (that is, it becomes of kind simple) if it has non-universal type. If it has universal type it retains its kind.
3) A STRING subscript is not really a subscript, it only specifies a substring. It is the same as specifying a subpartition of an ARRAY.

Consider the following example:

```
SET V(3,A,B), S1 $
    LX,U    B9,V
    LMJ     B11, 09SET $
    +       3
```

```
      G       SIMPLE$, INTEGER$, CONSTANT$, (3)
      G       040,0,NAME$,A
      G       040,0,NAME$,B
      'V     '
      LX,U    B9,S1
      LMJ     B11,09SET$
      +       0
      'S1    '

BOOLEAN CLASS TOPOLOGY (V..'TOP') $
      LX,U    B9, TOPOLOGY
      LMJ     B11, 09CLA$
      +       2
      G       043,0 NAME$, V
      G       SIMPLE$, STRING$, NAME$, (F LENGTH,,('TØP'))
      'TOPOLO'

TOPOLOGY(7..A) = V + B LSS TOPOLOGY$
      LX,U    B9, TOPOLOGY
      LMJ     B11, 08SSC$
      +       2
      G       SIMPLE$, INTEGER$, CONSTANT$, (7)
      G       040,0,NAME$,A
      SX      B9, TEMP$(1), B10
      LMJ     B11, 08ADD$
      G       043,0,NAME$,V
      G       040,0,NAME$,B
      SA      A2, TEMP$(2), B10
      LMJ     B11, 08LSS$
      G       043, RESULT$,B10,TEMP$(2)
      G       040, BOOLEAN$ + NAME$,, TOPOLOGY
      SA      A2,*TEMP$(1),B10
```

OPERATIONS INVOLVING GENERALIZED VARIABLES

Generalized variables can be used in arithmetic, Boolean, and relational operations. The ability to do this is taken care of by the operation table. The operation table is constructed as follows:

line 1	+	P
line 2	+	'<OP_1>',ad_1
to		:
P+1	+	'<OP_p>',ad_p

where P is the number of operations that are in the table, '<OP_i>' is the fieldata of one of the allowed operations, and ad_i is the address of the routine that performs the operation given by '<OP_i>'.

The compiler generates calls to routines 08ADD$, 08SUB$, 08OR$, etc. which are in the standard Algol library. These routines select the address of the operation table by examining the descriptors of the left and right operands. If the left operand is a non-simple variable and there is an address in the H1 field of the pointer (i.e. H1 of the pointer $\neq$ 0), the library routines use this address as the location of the operation table. If the above test fails for the left operand, the right operand is then tested (only one operand is tested for a unary operation). If neither operand has an operation table associated with it, an error message will be generated. Once the operation table is found, the library routine searches the operation table for the operation (e.g., 08ADD$ searches for 'ADD'). If the operation is found, the library routine transfers control to the user's routine to perform the operation. If the operation is not found, an appropriate error message will be generated. When the user is finished, he leaves the result in the proper registers and returns to the Algol program via the address in 08XIT$.

The general form for an operation is (only one descriptor for unary operators):

```
LMJ     B11,08---$
        <descriptor for left operand>
        <descriptor for right operand>
```

The general form for a subscript is:

```
LX,U    B9,<name>
LMJ     B11,08SSC$
+       n
```

<descriptor$_1$>
:
<descriptor$_n$>

Note that the user need not perform operations by use of the operation table. The user may call his routines 08ADD$, etc. and have control transferred directly to his own routines. However, when many different generalized variables are used in one program, each user operation routine would have to determine which operation to perform for the kind presented. The library routines decide which operation to call by looking at an operation table, and each different kind of generalized variable can have a different operation table.

GENERALIZED VARIABLE TRANSFER FUNCTIONS

It was mentioned before that each <classification$_i$> becomes a transfer function. The compiler generates the following coding:

```
LMJ                B11,07---$
<descriptor for parameter>
```

Consider the following example:

```
S1 = SET(A)
     LMJ           B11,07SET$
     +             1
     G             040,0,,A
     SA            A2,TEMP$(2),B10
     LMJ           B11,08REP$
     G             043,0,,S1
     G             043,RESULT$,B10,TEMP$(2)
```

REGISTER USAGE

The registers that must not be tampered with in any way by user routines are B1, B2, B3, B4, and B10. Failure to observe this rule will yield catastrophic results.

ACQUIRING AND RELEASING OF CORE

For most generalized variables, core will be needed either for the variable or for information relating to the variable. Core, however, may

not be used indiscriminately. To request a block of core:

1) Load A3 with the number of words desired.
2) Make the call: SLJ OPRO$
3) The address of the start of the block of the size requested is in AØ upon return from OPRO$.

If no blcok of the requested size is available then control is transferred to the library error routine. If transfer to the error routine is not desired then the following call should be used: SLJ OEPRO$. In this case if the block is not available the return is made with the contents of register A0 = 0.

Care must be taken to see that no core is used except exactly that which has been requested. When finished with the core, it may be "put back" for future use by:

1) Loading AØ with the starting address of the core to be released.
2) Make the call: SLJ OCON$

Only the exact block of core which was requested may be returned. Both OPRO$ and OCON$ destroy registers AØ, A1, A2, A3, and A4.

THE DEALLOCATION LIST

The deallocation list is a list that contains those items that should be deallocated when leaving a block or a procedure. There is a separate list associated with each block or procedure. To reference the head of the allocation list anything equivalent to the following form is acceptable.

OHEAD$,B10,H2

If the list head is zero then the list is null.

Items are placed on the deallocation list in LIFO ordered by the library utility routines OINS$ and removed by OREM$.

The contents of a list item can be one of two forms:

1) The H2 field contains an address in which case it is assumed to point to a block which is to be deallocated. OCON$ is called with this address as a parameter. The H1 field is ignored.
2) The H1 field contains an address and the H2 field contains a negative parameter which is discussed below. The address is assumed to point to a subroutine that

accepts the positive parameter as input. The subroutine is assumed to handle all deallocation that is necessary.

Entry into the subroutine is made in the following manner:

```
LM     A0, -<parameter>,*B0,XU
LMJ    B6,<subroutine>
```

The registers which cannot be used unless saved and restored are B1,B2,B3,B4,B8,B10,A7

A FINAL WORD

With generalized variables the user is able to declare and manipulate otherwise unwieldy variables. Many sophistications are possible and many problems are also possible. But, believe it or not, it can be made to work. OR rather, you can be made to work it.

APPENDIX IV

SORT-MERGE PACKAGE

Sort-merge facilities operating within the Algol framework are now available. The basic units with which the sort-merge package works are files. A file is defined as an area of storage in core, on drum, or on magnetic tape. Thus there are three types of files, namely: core, drum, and tape files. A file is well defined whether or not it has any relevant information stored in it. The creation (i.e. definition) of files is done using the Algol DEFINE declaration, and they may be processed, written, or read only by the sort-merge package. If a file has been written into, the items are in the form of Algol strings, which may be of different lengths. A summary of allowable calling sequences follows:

FILES:

```
DEFINE FILE $
FILE <identifier>('<type>',<length>) $
```

where <type> is either CORE, DRUM, or TAPE, and, for types CORE and DRUM, <length> is the file length in words. For type TAPE, <length> is a list of the form <'<logical unit>', <usable tape length in feet>,'<logical unit>', <usable tape length in feet>, ...>. The file is assumed to progress through the tapes listed in the declaration from left to right. A tape swap may be called for by following <length> by ,'SWAP'.

EXAMPLES:

```
DEFINE FILE $
FILE F1('CORE',400),B('TAPE','A',1200,'B',600),
     DRUMFILE('DRUM',5000),F2('CORE',800),
     SUPER('TAPE','C',1200,'D',1200,'SWAP')$
```

Here, two areas of core of lengths 400 and 800 words are set aside for core files F1 and F2. A 5000 word area of drum space is cleared and set aside for drum file DRUMFILE. A tape file (B) of total length 1800 feet is assumed to exist starting with 1200 feet of tape on logical unit A with an additional 600 feet of space on unit B. A tape file of undetermined length is assumed to exist with an undetermined number of 1200 foot tapes on logical units C and D, starting with 1200 feet on unit C continuing on unit D, and returning to C, moving cyclically between units C and D.

Note that, as with Algol 60 arrays, when storage is declared (i.e. either a file or an array declaration), the space declared is assumed empty, and remains empty until the user places items into the file or array. Only a previously created non-empty tape file may be used as a tape file input in a new program. Also, it should be noted that the Algol drum routine and the sort-merge drum files are not compatible and hence exclusive use of one or the other of these drum processing methods should be made.

SORTKEYS:

DEFINE SORTKEY$

SORTKEY <identifier>(<start char$_1$>,<char length$_1$>,...,
<start char$_N$>,<char length$_N$>) $

where <start char$_I$> is the starting character position and <char length$_I$> is the character length of the Ith most significant key.

EXAMPLES:

```
DEFINE SORTKEY$
SORTKEY KY(20,4,6,8),DATAKY(1,5,10,5),
        SPECIAL(1,80),VACATION(27,2,25,2,1,20) $
```

Suppose that a file, FIL, contains information. In particular, suppose that for each item in FIL, characters 1-20 designate an employee name, characters 25-26 and 27-28 contain the day and month (01-12, where 01 is January), respectively, of the start of the employee's vacation period. A subsequent sort of file FIL with the key VACATION as defined above would produce a group of items grouped by vacation date with the vacation dates in chronological order. If more than one employee starts his vacation on a given date, the items are grouped in alphabetic order by employee name. Note that a key is meaningful only with respect to a sort or a merge.

POOLS:

DEFINE POOL$

POOL <identifier>(L) $

where L is as defined for tape files, without the swap option. Pools must consist of magnetic tape, and are only used for temporary storage by a sort process.

EXAMPLES:

```
DEFINE POOL$
POOL SPARE ('M',3600,'P',1200,'R',300),
     ROOM('N',1200,'O',2400) $
```

Here, SPARE provides 5100 feet of magnetic tape for intermediate

storage for the system during a sort operation. Note that a pool is used only for intermediate storage in a sort operation.

Sort-merge procedure calls:

SORT(<output>,<input>,<key identifier>)

SORT(<output>,<input>,<key identifier>,<pool identifier>)

MERGE(<output>,<input>$_1$,<input>$_2$,...,<input>$_n$,<key identifier>)

TRANSFER(<output>,<input>)

TRANSFER(<output>,<input>,<intermediate procedure>)

TRANSFER(<output>,<input>,<boolean>)

TRANSFER(<output>,<input>,<intermediate procedure>,<boolean>)

where <output> is either 'NONE', a file name, or the name of an Algol procedure having a single string parameter; where <input> is either a file name, an Algol string procedure name with no parameters, a call on SORT, a call on MERGE, or a call on TRANSFER; where <intermediate procedure> is an Algol string procedure having a single string parameter; and where <boolean> is either a boolean variable or an Algol boolean procedure.

EXAMPLES:

Suppose it is desired to load items from cards into a file, F. Let the items be arranged on cards so that one item appears per card and so that the item is within columns 1-40. A program which accomplishes this is shown below:

```
 1      DEFINE FILE $
 2      FILE F('TAPE','A',1100) $
 3      STRING PROCEDURE IN $
 4           BEGIN OWN STRING S(40) $
 5           FORMAT FORM (A,S40) $
 6           READ (FORM,S,STOP) $
 7           IN = S $ GOTO OUT $
 8 STOP..    IN = '' $
 9 OUT..     END IN $
10      TRANSFER (F,IN)
```

Note that, as shown on line 8, the TRANSFER operation is terminated by setting the input equal to a null string (IN = ''). A null string is defined to be a string of character length zero, represented by two adjacent quote marks. (See the flowcharts at the end of this appendix.)

Assume now that we desire to delete certain items from file F, along with deleting some of the item information. In particular, suppose that a "D"

in the first character of an image indicates that that item is to be deleted. Also, columns 2-4 are to be set to ØØØ for all items. A program which accomplishes this is shown below.

```
1       DEFINE FILE $
2       FILE FNEW('TAPE','B',1200),F('TAPE','A',1100) $
3       STRING PROCEDURE EDIT(S) $ STRING S $
4            BEGIN IF S EQL " THEN EDIT = " ELSE
                   IF S(1) EQL 'D' THEN EDIT = "
5            ELSE BEGIN S(2,3) = 'ØØØ'$
6                 EDIT = S END
7            END EDIT $
8       TRANSFER (FNEW,F,EDIT)
```

Here, an item is taken from the input, F, and passed into the intermediate procedure EDIT. Note that EDIT must check for an "end of file" (null string - line 4) from the input. If an item is to be deleted, the output of the intermediate procedure EDIT is set to a null string (line 4). Note that a null string from an intermediate procedure is not passed on to the output and operation continues, while a null string from the input serves as an end of file, and operation terminates after passing the item to the output. See the flowcharts in this appendix. Note that on line 4 S is checked for the null string first. Otherwise, if S is null and S(1) is checked against 'D' first the library will consider this an error.

Now, as a check, let's print the first 100 items of file FNEW. An appropriate program is shown below.

```
1       INTEGER I $
2       DEFINE FILE $
3       BOOLEAN BOOL $
4       STRING PROCEDURE COUNT(S) $ STRING S $
5            BEGIN IF I EQL 101 OR S EQL '' THEN
6                  BEGIN BOOL = TRUE $ COUNT = '' $
7                  GOTO EXIT END
8            ELSE I = I + 1 $ COUNT = S $
9  EXIT..    END COUNT $
10      PROCEDURE OUT(S) $ STRING S $
11           BEGIN IF S NEQ '' THEN WRITE (S) END OUT $
12      FILE FNEW('TAPE','B',1100) $
13      I = 1 $ BOOL = FALSE $
14      TRANSFER (OUT,FNEW,COUNT,BOOL)
```

Here, after 100 items have been encountered or if an "end of file" has occurred, a boolean variable is set to TRUE (line 6). With the boolean variable set to TRUE, the process is terminated by setting the intermediate result to a null string. See the flowcharts for more detail.

Now, let's sort the file FNEW and place the sorted file in file F. In sorting, items will be sorted by 4 characters starting at character 10, and by 2 characters starting at character 20. The additional coding necessary to accomplish this is shown below.

```
1     DEFINE SORTKEY $
2     SORTKEY KY(10,4,20,2) $
3     FILE F('TAPE','A',1100) $
4     SORT (F,FNEW,KY)
```

Note that sorting and merging produce output in ascending order.

If we wanted to merge the (sorted) file F with, for example, items from a sorted file EXTRA, placing the entire output on tape, the call would appear as (using the same key as above).

```
MERGE (TOUT,FNEW,EXTRA,KY)
```

where TOUT is an (tape) output procedure constructed as before.

If equal minimum items occur in merging, the left-most input in the merge call is taken as the actual input, that is, with respect to the calling sequence of a merge, the item processed at each pass is the "left-most" minimum item.

VOLATILE DRUM SPACE

It should be noted by the user that any DRUM file or any call on the SORT routine may destroy part of the user's PCF and/or the "drum tape" region used by CUR and Algol. Thus, if information is required to be in the PCF or "drum tape" region it must be saved by the user, elsewhere than on drum.

AVAILABLE DRUM SPACE

The number of words of drum storage available to the user's program is a dynamic property of the operating system. That is, it is subject to change as the needs of system change.

In general, the SORT routine requires temporary storage approximately equal to the total length of the final set of sorted items. Therefore, if the sorted items are to be kept in a drum file, it will not be necessary to have POOL tapes if only about one-half of the available drum is used for a drum file, leaving the rest for temporary storage for the SORT routine.

FILES

Upon declaration, all files are set closed and empty (i.e., the file contains only a null element). Let A be a properly defined S/M file. Let C be defined by

FILE C$ C=A

If A is not a tape file, define file B as

FILE B(A)$

Any operation with file C is equivalent to performing that operation with file A. File B, on the other hand, operates as an independent file using the same storage area. Thus with files of the A and B type, informatio may be written into and read from the same storage area simultaneously. That is, using files A and B, two (possibly different) types of operations may be performed on a given set of items simultaneously.

Thus, an operation of the form

TRANSFER(A,A,INTER) or MERGE(A,A,B,KY),

where the previous definitions for A and B hold, is well defined.*

SUBSCRIPTED FILES, SORTKEYS AND POOLS

In processing information it is often necessary to perform the same operation or sets of operations on several different sets of data. In Algol 60 the FOR statement is ideal for circulating through a given set of operations a known number of times.

For files, sortkeys and pools, if A has been previously declared and is of a corresponding type, declarations of the form

* The user should note that an operation of the form

SORT(A,A,KY)

is well defined since file A is opened, read, and closed before items are written into file A.

1) <declarator> B(A),C,D(<integer>)

and, in addition, in the case of a file declaration,

2) FILE E(<integer>,'<type>',...)

are allowed where <declarator> is either FILE, SORTKEY, or POOL. As with files, a replacement of the form

C=A

simply associates the name C with A. Any operation performed on C is identical to performing the same operation on A. In the case of sortkeys and pools, however, the declaration

<declarator> B(A)

does not have any special advantage, but merely duplicates the declaration used for A. That is, the effect of a declaration of the form

<declarator> B(A) $

for sortkeys and pools has the net effect of a declaration of the form

<declarator> B$

followed by B=A with the exception that internal storage is used in the former.case. Thus the latter type of declaration is preferable in associating different pool and sortkey names. The former type of declaration is only made available to maintain consistency.

The declaration

<declarator> D(<integer>)

is used to denote a subscripted file, sortkey or pool whose subscripts will range from 1 to <integer>, where (<integer>)>ø.

Actual elements are associated with the subscripted file, sortkey, or pool by replacement statements of the form

D(<integer>) = A

where A is some (possibly subscripted) file, sortkey, or pool.

The special file declaration simply declares <integer> files, all of the same type, with subscripts ranging from 1 to <integer>, (<integer>)>ø.

MULTIPLE FILE ACCESS

In order to incorporate virtually complete flexibility within the Sort/Merge framework it is necessary to enable the user to read/write items from/to

several files within the Algol 60 framework. Thus the S/M package will accept statements of the form:

<file name> = <string>

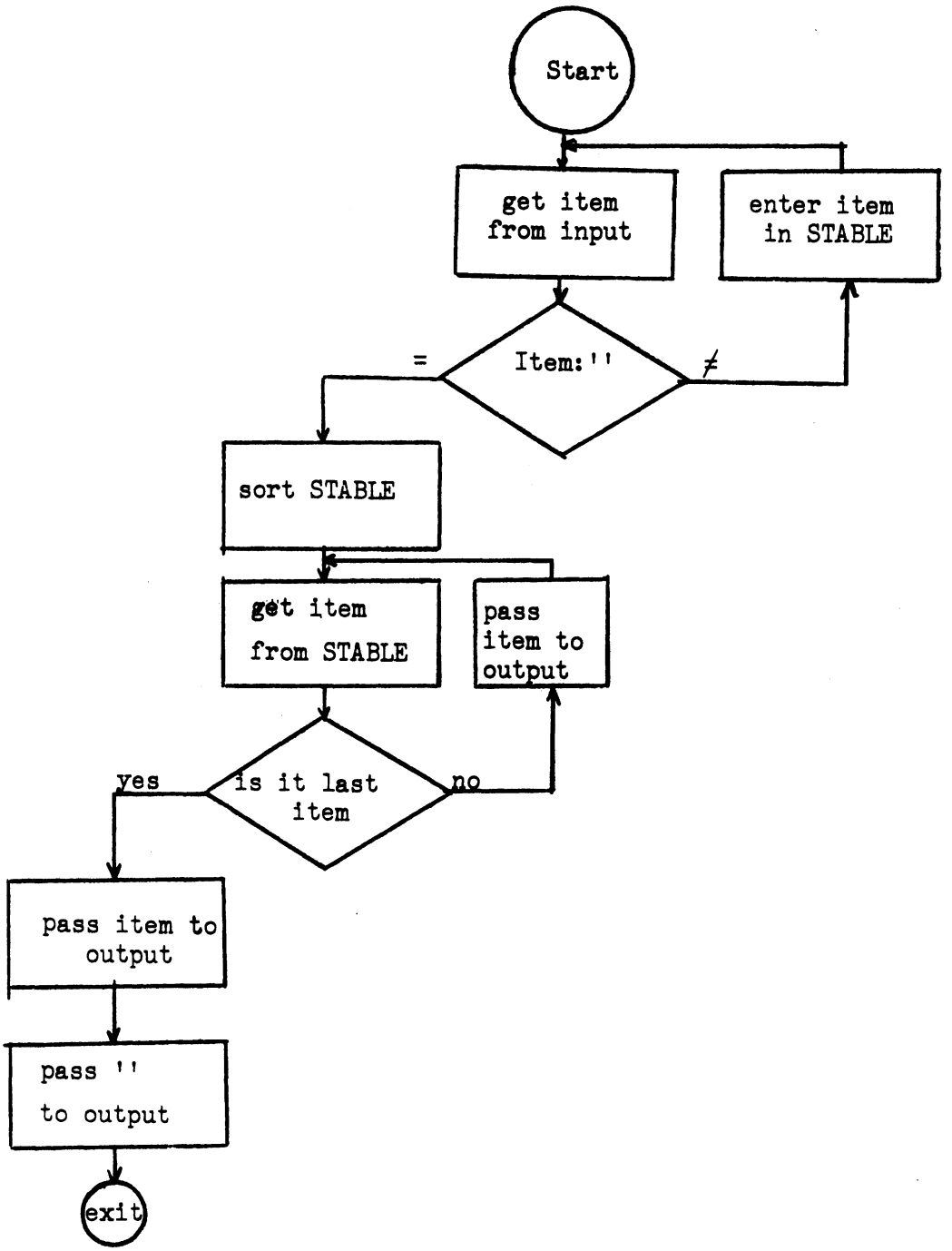
or

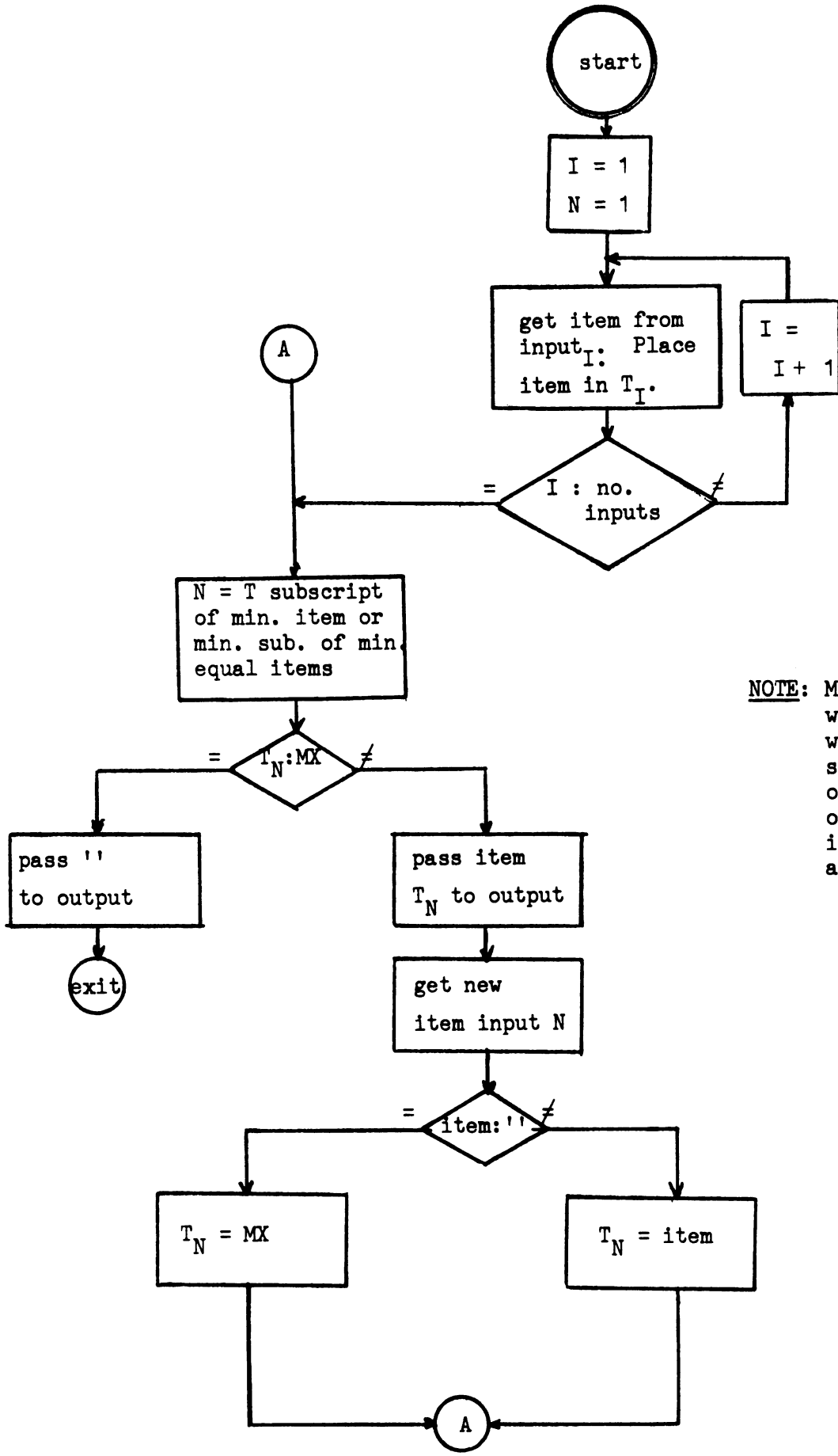
<string name> = <file name>

where the first form can be interpreted to mean "write the item whose representation is <string> into file <file name>", with a corresponding interpretation of the second form of "copy next item from file <file name> into <string name>". Note that no limitation is made on the number of different files used, but that the file read/write rules described previously still apply. The only additional restriction that applies here is that these statements must be executed <u>during</u> a call on the S/M package, i.e., in a procedure currently being called by the S/M package.

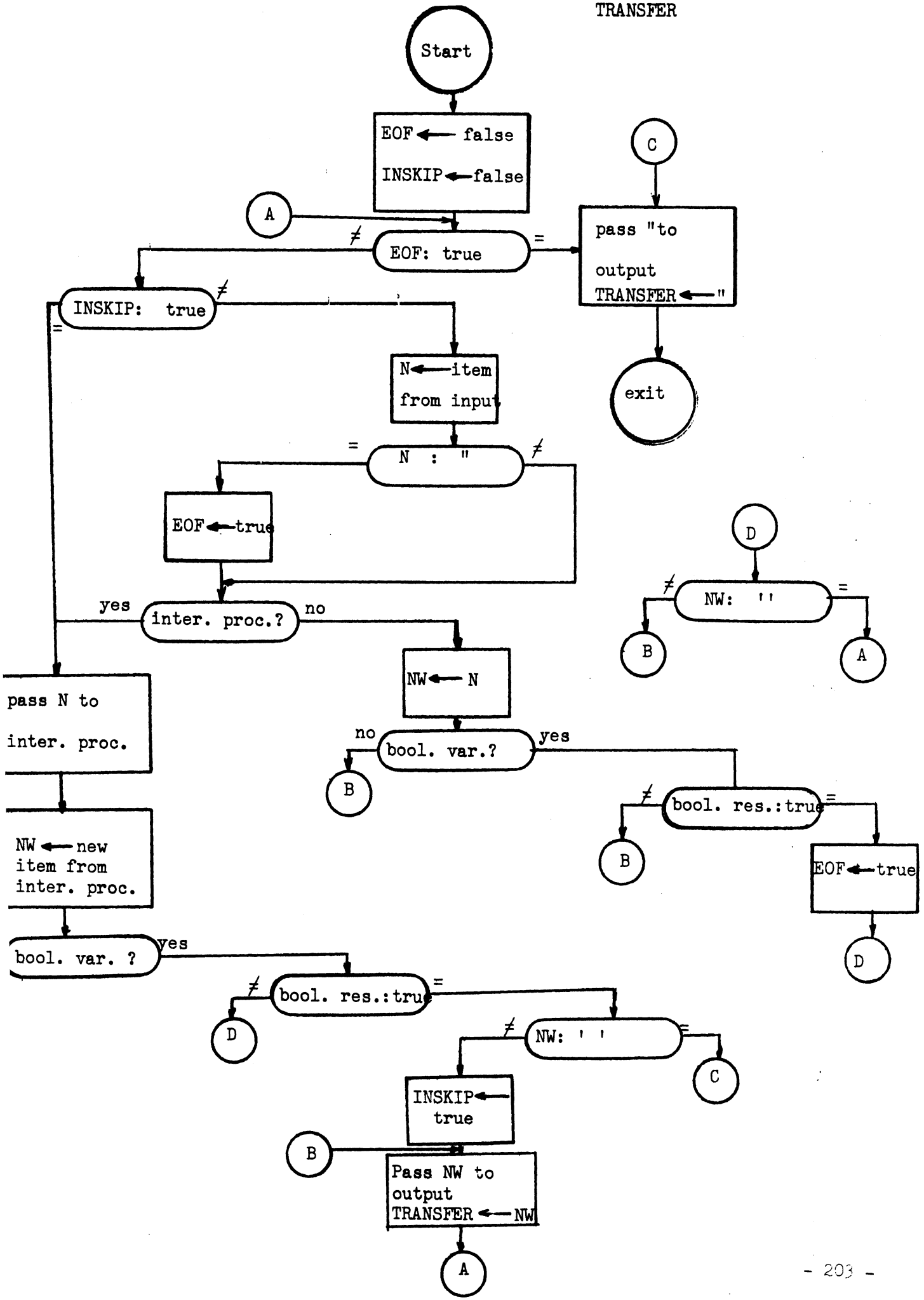
NOTE: Ascending order is determined by the result of a <u>SETRANK</u> statement, if one or more occur in the program.

FLOWCHARTS
SORT





NOTE: MX is an item which when sorted along with all possible string combinations of symbols, would occur as the last item in the sorted ascending sequence.


Start
EOF ← false
INSKIP ← false
A
EOF: true
≠
=
C
pass "to
output
TRANSFER ← "
exit
INSKIP: true
≠
=
N ← item
from input
N : "
=
≠
EOF ← true
yes
inter. proc.?
no
pass N to
inter. proc.
NW ← new
item from
inter. proc.
bool. var. ?
yes
NW ← N
no
bool. var.?
yes
B
D
NW: ' '
≠
=
A
≠
bool. res.: true
=
EOF ← true
D
≠
bool. res.: true
=
D
≠
NW: ' '
=
C
INSKIP ←
true
B
Pass NW to
output
TRANSFER ← NW
A

APPENDIX V

PLOTTER ROUTINES

THE CALCOMP PLOTTER

The Calcomp 565 plotter uses a roll of paper 120 feet long and 12 inches wide. The actual plotting width is 11 inches. A rotating drum moves the paper lengthwise, while a carriage moves a replaceable pen back and forth along the width of the paper. Both ball point and ink pens are available.

The pen-up-pen-down functions control writing. Motion is in 0.005 inch increments at a maximum rate of 300 steps per second. Drawings, therefore, consist of short line segments oriented in increments of 45 degrees. Repositioning to a reference point is precise. A user may run a drawing 120 feet long and then accurately reposition at the point of origin.

The control unit and the plotter must both be on (red indicator lamps on control unit) to operate the plotter. Also, the stop lamp (white) must be extinguished manually, if it is on, by depressing "run". Depressing "stop" turns on the stop lamp and stops operation until "run" is depressed. If the computer issues a stop command, the effect is the same as a manual stop.

Manual controls on the plotter are used to remove completed drawings.

BASIC PLOTTER OPERATION

The various routines needed for use of the plotter will compute the necessary control codes and write them to a magnetic tape. This tape must be assigned to logical unit 'P' by means of an ASG control card.

After a program has written plotting instructions on the tape, the graph can be plotted by means of an EXEC III parasite called PLT. To initiate the plotting parasite the user makes an unsolicited keyin of the form:

I PLT <label>,<paper feed>

where <label> is the label of a tape to be plotted. This label must have been previously assigned to a physical unit by means of an unsolicited keyin of the form:

A <label>/<physical unit>

<paper feed> is the distance which the paper will be advanced through the plotter in units of inches/100 before starting the plot. For example, the keyin:

I PLT GRAPH, 300

will initiate the plotter parasite. The parasite will plot the information contained on the tape on the physical unit assigned to the label called GRAPH. Before plotting, PLT will advance the paper 3 inches.

PLT may be suspended or terminated by the usual unsolicited keyins. The tape is not rewound after plotting.

An EOF marker may be written on the tape using Algol or CUR. This will be considered a stop command by the PLT parasite. With the use of CUR a specific plot may be found on a tape and then plotted with the PLT parasite.

ALGOL PLOTTER PROCEDURES

There are several procedures available for use from an Algol program to produce output for the plotter. These are available in the system stored library on the drum.

The available procedures are as follows:

1. SCALE (<initial x>,<initial y>,<x scale>,<y scale>,<max y>,<grid type>, <x unit>,<y unit>,<coordinate system>)

   a). Sets the initial point (see figure 3) to correspond to the scaled coordinates:

   (<initial x>,<initial y>)

   where <initial x> and <initial y> must be real numbers.

   b). Sets the scale of the x axis to <x scale> units/inch. <x scale> must be a real number, but may be positive or negative.

   c). Sets the scale of the y axis to <y scale> units/inch. <y scale> must be a real number, but may be positive or negative.

   d). Sets the maximum expected y coordinate to be <max y>.

   e). Draws a grid indicated by <grid type>, which must be of type STRING.

   i). If <grid type> is 'Δ' then no grid is drawn.

   ii). If <grid type > is 'AXIS' then an x-y coordinate axis is

DIAGRAM OF PLOTTER

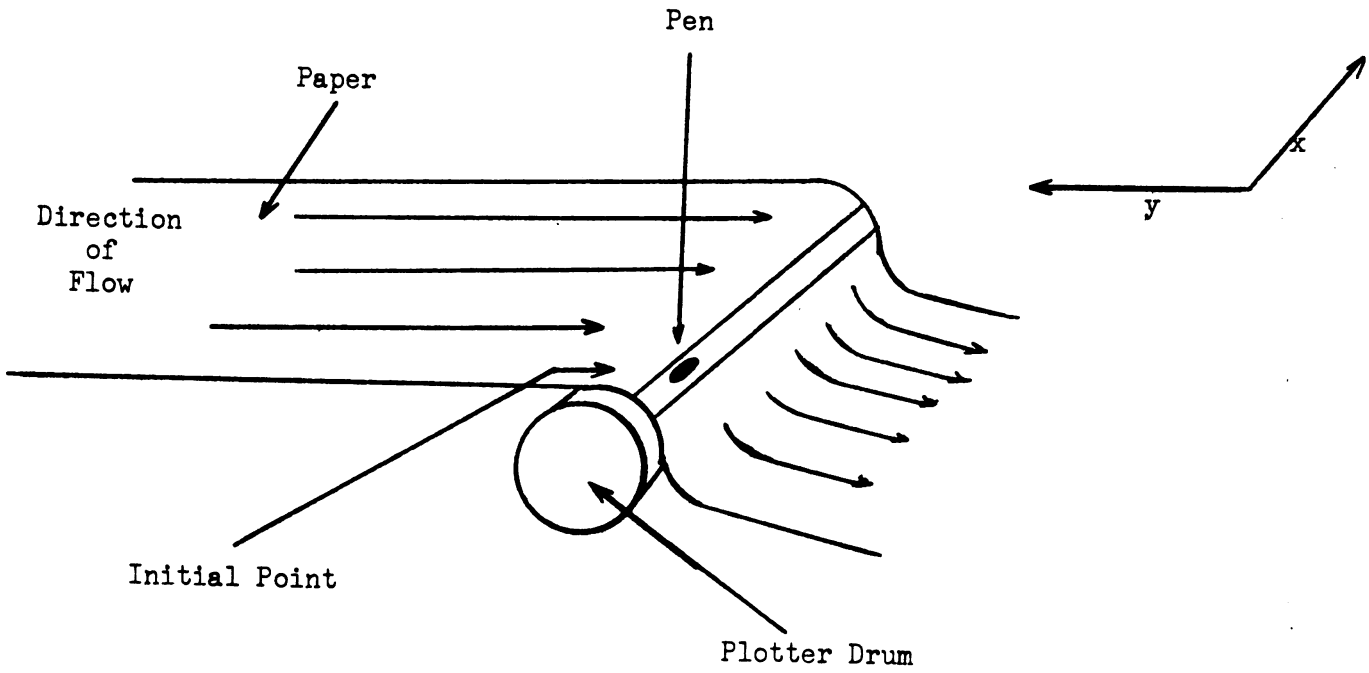


Figure 3

drawn the width of the paper, and from <initial y> to <max y>. Marks are drawn on the axes in increments of <x unit> and <y unit> for the x and y axes. The origin of the coordinate axes is the scaled point (0.0,0.0). <x unit> and <y unit> must be real numbers.

iii). If <grid type> is 'GRID' then a grid is drawn with lines crossing at intervals of <x unit> units in the x direction and <y unit> units in the y direction. <x unit> and <y unit> must be real numbers.

f). If <coordinate system> is 'POLAR' then all subsequent plotting (until the next call on SCALE) will be in polar coordinates. If <coordinate system> is 'CARTESIAN' then Cartesian coordinates will be used. If this parameter is omitted from the call on SCALE then Cartesian coordinates will be used.

g. Only in the case of <grid type> equal to 'Δ' is the pen positioned to the initial point. In the other cases the pen may be anywhere on the paper.

2. PLOT(<x coord.>, <y coord.>, <line type>, <arrival action>)

a). Puts the pen down

b). If plotting is being done in polar coordinates then <x coord.> is the radius, and <y coord.> is the angle in radians of the point to which the pen is to be moved, where <x coord.> and <y coord.> are real numbers.

c). If plotting is being done in Cartesian coordinates then the pen is moved to the point (<x coord.>, <y coord.>).

d). If the string <line type > is 'DASH' then a dashed line is drawn, while a continuous line is drawn if <line type> is 'SOLID'.

e). The <arrival action> parameter is of type STRING. Its value should be one of the following:

i). 'BOX' draw a small box around the point (<x coord.>, <y coord.>)

ii). 'CROSS' draw a small line perpendicular to the direction of travel at the point (<x coord.>, <y coord.>)

iii). 'ARROW' draw an arrow head in the direction of travel, with the point of the arrow head at (<x coord.>, <y coord.>)

iv). 'Δ' take no action at the end of the line.

3. MOVE (<x coord.>, <y coord.>)

   a). raise the pen

   b). moves the pen to the point (<x coord.>, <y coord.>) if the plotting is in Cartesian coordinates, or uses <x coord.> as the radius, and <y coord.> as the angle in radians of the point to move to, if the plotting is in polar coordinates. <x coord.> and <y coord.> must be real numbers.

4. CHAR (<direction>, <size>, <string>)

   a). puts the pen down

   b). draws the characters of <string> with a height of <size> inches, in the direction of <direction>, where the present pen position is the lower left-hand corner of the first character.

   c). if <direction> is of type STRING then its value should be one of the following

      i). 'X' edit the string in the positive x direction

      ii). '-X' edit the string in the negative x direction

      iii). 'Y' edit the string in the positive y direction

      iv). '-Y' edit the string in the negative y direction

   d). If <direction> is not of type STRING then it should be of type REAL, in which case it is the angle in degrees at which the string is to be plotted. The angle is measured counterclockwise, with the positive x axis denoted by an angle of zero degrees.

   e). The point at which the pen is positioned when the CHAR operation is done is described in 11) of the MISCELLANEOUS NOTES section of this appendix.

5. SCHAR (<direction>, <size>, <index>)

   a). SCHAR is similar to CHAR except that SCHAR draws only one character. This character is selected from the table of character indices according to the value of <index>, which must be of type INTEGER. The table and ways to add or delete characters are explained later in this appendix.

   b). The point at which the pen is positioned when the SCHAR operation is done is described in 11) of the MISCELLANEOUS NOTES section of this appendix.

6. STOP

   causes the generation of a stop command by the computer. The user must manually release the plotter to continue plotting.

7. NEWGRAPH(<inches>)

moves the pen <inches> beyond the point (<initial x>,<y max>) given by the previous SCALE call. <inches> should be of type REAL.

8. DPARAM(<length>,<spacing>)

changes the length and spacing of dashed lines so that the value of <length> is the length of the dash and the value of <spacing> is the size of the gap between dashes. Both parameters should be of type REAL and are in units of inches.

9. KEYCOUNT

is an INTEGER PROCEDURE without parameters. Its value is the number of characters that can be indexed by SCHAR (at the time of this writing the value is 168).

10. CHANGEKEY (<index>,<pair list>)

is used to add or delete characters in the table of character indices. <index>, a parameter of type INTEGER, should be such that

$$\emptyset \leq |\langle\text{index}\rangle| \leq \text{KEYCOUNT} + 19$$

<pair list> is a sequence of pairs of INTEGER parameters that describe the new character.

If a character with index |<index>| is already present, it will be deleted. The new character is then added. The <pair list> is a sequence of x-y coordinates. Each <integer expression> must be in the range $-31 \leq$ <integer expression> $\leq +32$. The pen will be raised before starting each character. If the x coordinate is +32 then the pen will be raised, and the y coordinate is ignored. It will be dropped after performing the next pen movement. As an example of usage, the call

CHANGEKEY(KEYCOUNT, Ø,5, 4,5, 32,Ø, 4,3, Ø,3)

would put an equal sign in the character set, and remove the character (if any) which had index KEYCOUNT. It should be noted that a call on CHANGEKEY does not change the value of KEYCOUNT.

If <index> is negative, then the pen will stay at the last point it was moved to. An example of this usage is the backspace character, which is defined by the call CHANGEKEY(-KEYCOUNT, -6,Ø)

FORMATTED OUTPUT ON THE PLOTTER

The use of the EDIT input/output device in conjunction with the plotter is illustrated in Appendix VI.

TABLE OF CHARACTER INDICES

The following characters are now available:

	Ø	1	2	3	4	5	6	7	8	9
Ø	∇	[	]	#	Δ		A	B	C	D
1	E	F	G	H	I	J	K	L	M	N
2	O	P	Q	R	S	T	U	V	W	X
3	Y	Z	)	-	+	<	=	>	+	$
4	*	(	%	:	?	!	,	\	Ø	1
5	2	3	4	5	6	7	8	9	'	;
6	/	.	□	≠	→	←	"	'	'	
7	a	b	c	d	e	f	g	h	i	j
8	k	l	m	n	o	p	q	r	s	t
9	u	v	w	x	y	z	✱	◇		
1Ø	A	B	Γ	Δ	E	Z	H	Θ	I	K
11	Λ	M	N	Ξ	Ω	Π	P	Σ	T	ϒ
12	Φ	χ	Ψ	ω	α	β	γ	δ	ε	ζ
13	η	θ	ι	κ	λ	μ	ν	ξ	o	π
14	ρ	σ	τ	υ	φ	χ	ψ	ω		
15	⊂	'	|	∋	≤	≥	∂	↑	↓	¬
16	∧	⊃	≡	∀	∃					

Those with an index greater than 63 can be used only with SCHAR. Note the following characters:

index	5:	blank
	69:	backspace
	98:	under dash
	99:	under line
	148:	over dash
	149:	over line
	165:	superscript shift
	166:	subscript shift

If the character size for 165 and 166 is the same, they will return to the original vertical position on the line. For example:

```
SCHAR('X',Ø.1Ø,74) $ SCHAR('X',Ø.10,165) $
SCHAR('X',Ø.1Ø,93) $ SCHAR('X',Ø.1Ø,166) $
CHAR('X',Ø.1Ø,'+1')
```

will plot $e^{x}+1$.

MISCELLANEOUS NOTES

1). Improper usage of the plotter routines can bring swift retribution as there is no error checking with respect to the parameters.

2). The user deserves what he gets if he uses any plotter routines before the first call on SCALE.

3). Note the action taken at the boundaries by the plotter routines. A sample might be:

```
SCALE (0.0,0.0,1.0,1.0,' ',0.5,0.5) $ MOVE(10.0,10.0) $
PLOT (12.0,12.0,'SOLID',' ')
```

This is equivalent, with the same SCALE call, to

```
MOVE(10.0,10.0) $ PLOT(11.0,11.0,'SOLID',' ') $ PLOT(11.0,12.0,'SOLID',' ')
```

Note also, in this example, that it is possible to plot beyond YMAX without ill effects.

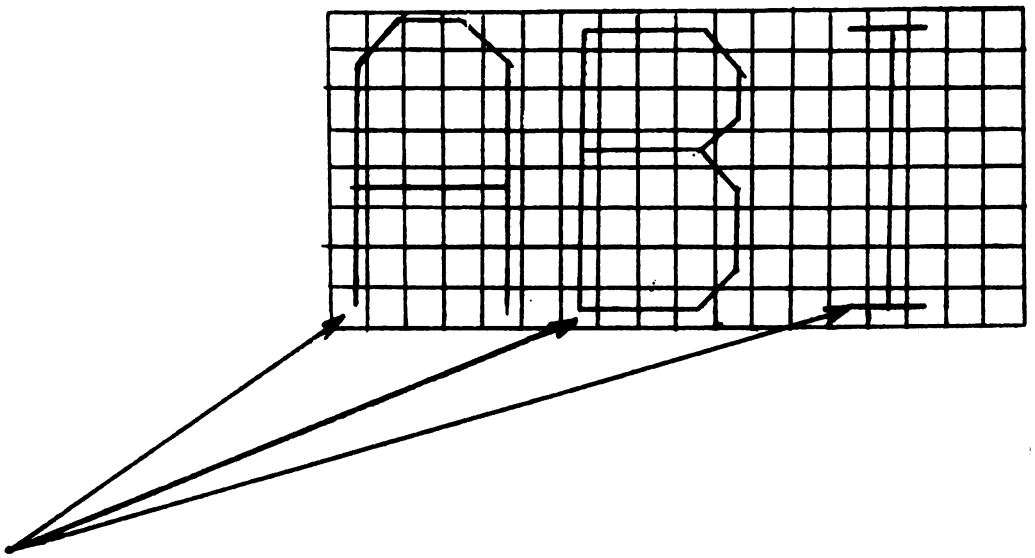
4). Plotter movements of more than 109.22666 feet in the y direction (on one call to the plotter routines), may produce anomalous results.

5). When using ink, the user should impulse the pen up and down several times to start the ink flowing. This is best done manually.

6). It is possible to plot two (or more) graphs, in different scales, on top of each other. This is done by omitting the call on NEWGRAPH between plots, and positioning the pen so that the point on the left-hand boundary of the paper opposite the pen is the desired initial point. It is not necessary to actually move the pen to the initial point, since the call on SCALE will do this. For example, if the pen is presently at (x,y), then the initial point after the next call on SCALE corresponds to the point (<initial x>, y) under the present SCALE.

7). To guarantee that the last buffer will be written to tape, the user should execute a STOP at the end of his program. Programs terminated because of errors may or may not have the last buffer written to tape.

8). BASIC CHARACTER FORM



9). Upon entering CHAR or SCHAR, pen is assumed to be positioned at indicated point.

10). The basic character grid is eight units high, and five units wide. Each character starts six units from the start of the preceding character. The characters themselves are seven units high and four units wide.

11). After plotting a character the pen will be positioned in the lower left-hand corner of the next character grid except when the character just plotted is backspace (69), superscript shift (165) or subscript shift (166). Each of these three characters will leave the pen in the final pen position of that character and all others will leave the pen in the (0,0) position of the next character.

APPENDIX VI

SPECIAL INPUT/OUTPUT DEVICES:

EDIT, CORE, PCF, SLIP

This appendix describes several special I/O devices that may be used in addition to the standard devices CARDS, PRINTER, PUNCH, TAPE and DRUM.

PCF: Program Complex File
SLIP: Source Language Input Processor

Two new devices have been added for use from Algol READ and WRITE calls. These are PCF and SLIP. They are used to get card images from a symbolic element in the user's complex. PCF may be used in READ and WRITE, while SLIP may be used in READ only. They are called in a manner similar to DRUM or TAPE, for example, READ(PCF(...)...).

PCF may have one or zero parameters. If there is a parameter it may be of type integer or string. If the parameter is a string, then it is the name/version(cycle) of an element in the user's complex. It does not have to be left justified in the string, but can be preceded by spaces. If the parameter is an integer, then it must be in the range one to six, and is used to index the name/versions present on the Processor call card. For example, READ(PCF('HELLO/DERE'),...) is a legitimate call on the PCF device. If the processor call card is

```
▽ ACE    AAAAAA/US,AAAAAA/GERMAN,AAAAAA/JAPAN
```

then READ(PCF(2),...) would be a request for a card image from the element AAAAAA/GERMAN. If there is no parameter to PCF, then the element used on the previous call will be used.

The PCF device is used to read or write sequential card images to or from an element in the user's complex. The parameter indicates which element is to be read or written. Each request on the PCF device when reading will pass one card image (80 columns) from the complex to the read routine, to be processed according to the format included in the call, or the implied

format for the card reader if there is no format included in the call.

Similarly, each request on the PCF device when writing will pass one card image (80 columns) from the write routine to the complex, which has been prepared according to the format included in the call, or the implied format for the card punch if no format was included in the call. If more than 80 columns are passed to the PCF device only the first 80 will be used. All page ejects (E format phrases) are ignored. Line skips (Aw.d) are not ignored, but w = 0 is equivalent to w = 1.

The SLIP device is used to read a symbolic element from the user's complex, insert correction cards obtained from the card reader, and prepare an updated symbolic element, in a manner similar to the compiler. Please note that SLIP may not be used when calling WRITE. Furthermore, the results of calling WRITE(PCF(..)...) before an entire element has been SLIPped are unpredictable.

SLIP may have 0,1 or 2 parameters, of type integer or string. The first parameter to SLIP indicates the element to be updated, while the second parameter indicates the name of the updated symbolic. If the second parameter is not present, then there will be no updated symbolic entered into the user's complex. However, it is not possible in this case to write in the complex using the PCF device (while the SLIP device is being used). If no parameters are present, then the previous parameters will be used.

USE OF FORMATS WITH PCF AND SLIP

The PCF and SLIP devices will act just like the card reader or card punch. Thus, formats are optional, but generally desirable.

ERROR RETURNS FROM PCF AND SLIP

If the error label is present on a call to PCF or SLIP it will be used if any errors are found while processing the call.

The only type of abnormal return from PCF when reading is an EOF return when trying to read the (n+1)th card from an element containing only n cards. Any subsequent attempts to read the same element again will cause the element to be reread, i.e., the first time this happens card 1 will be processed, etc. Attempts to read an element which is not in the user's complex will cause the message '***** input source language element (XXXXXX/XXXXXX) not available' to be printed and the program will be terminated. It is, of course, possible for the read routine to discover errors, in which case the other error returns will be used.

There are no abnormal returns possible from PCF when writing. The only error which can occur will be insufficient space available on the drum (i.e., running off the end of the drum), and in this case the program will be terminated, with a message from the system.

The abnormal return from SLIP will depend on the type of control card present at the end of the correction cards being read in. The abnormal return will occur when an attempt is made to read the (n+1)th card of an n card element. If the correction cards are terminated with an EOF card, then a return will be made to the EOF label in the input list. Any other type of control card will cause a return to the EOI label. Further attempts to SLIP the same element will cause the element to be reread. Note that the program will be terminated if the appropriate error label is not present. If a third label is present in the input list then it will be used in case of a card read error by the Algol READ routine, or is case an error is found while SLIPping the element.

READING AND WRITING SEVERAL ELEMENTS AT ONCE WITH PCF

It is possible to be reading cards from several elements at once using PCF. The PCF device merely keeps track of all the elements being read which still have cards available. Each element being read uses 02002 octal words of memory (1026 decimal). It is possible to be reading SLIP and PCF at the same time, although they should be working on different elements. It is not possible to be writing several elements at the same time. Whenever the name/version (cycle) of the element being written is changed, the previous element is entered into the user's PCF. Any subsequent writing in the complex of an element of the same name will cause the first element to be deleted. That is, writing element A, and then B, and then A again, will cause the first element named A to be deleted.

The use of improper or incorrect correction cards with SLIP will cause the printing of one of the following messages: '***** ILLEGAL CORRECTION CARD', '***** CORRECTION CARD SEQUENCE ERROR', or '*****IMPROPER DELETION -mmmmm,nnnnn'. The correction card concerned will be ignored, and the

program will not be aborted.

Note that it is only possible to use integer parameters to PCF or SLIP when the program is being run as a Processor.

EXAMPLE OF THE USE OF SLIP

The following program is essentially the Processor DATA.

```
INTEGER I $ FORMAT F(A,S80), G(X10, I5, '.', X5, S80, A1) $
LOCAL LABEL EOF, FIN$ STRING A(80) $
FOR I = 1 STEP 1 UNTIL 10000000000 DO BEGIN
      READ(SLIP(1,2)F,A,EOF,FIN) $ WRITE(PRINTER,D,I,A) END $
EOF..FIN
```

At present, the maximum number of elements which may be entered into the complex with one call on XQT or a user's processor, is about 32.

THE CLOSE PROCEDURE

CLOSE is used to close out an input buffer being used by the PCF or SLIP device. A call on the CLOSE procedure has the form

CLOSE(<element name>)

where <element name> is of the same form as a parameter to the PCF or SLIP routine. The effect will be to delete the memory buffer being used by <element name> and return it to the available space pool.

Any subsequent attempts to access <element name> with the PCF or SLIP device will result in redefining the element and passing the first card(s) of the element. Since each buffer used by an element requires 02002 octal words of memory, this provides a means of releasing unused storage for other purposes.

THE PCFELT PROCEDURE

PCFELT is used to obtain information about the elements in the program complex file, regardless of their types. The call has the form

PCFELT(<string>,<type>,<oldest>,<newest>,<maximum>)

where the parameters are as follows:

<string> is the name of a STRING variable into which the PCFELT routine will place the name/version of the element. The characters are left-justified in the string, and a '/' (if needed) will separate the name from the version.

<type> is an INTEGER variable which is set equal to the type of the element.

<oldest> is an INTEGER variable that is set equal to the lowest cycle number available if the element is symbolic, and otherwise is set to zero.

<newest> is an INTEGER variable that is set equal to the highest cycle number available if the element is symbolic, and otherwise is set to zero.

<maximum> is an INTEGER variable that is set equal to the number of cycles that will be saved if the element is symbolic, and otherwise is set to zero.

The first call on PCFELT during execution of a program will return information about the first non-deleted element present in the complex. By first is meant the first element entered into the complex. The second call on PCFELT will return information about the second non-deleted element, and similarly, the Nth will deal with the Nth element. If there are M elements in the complex, the (M+1)th call on PCFELT will return with <type> set equal to zero and all other parameters unchanged. The (M+2)th call on PCFELT willstart from the beginning, i.e., will deal with the first element.

PCFELT operates on the entire complex, not just the symbolic elements. If the PCFELT and PCF-SLIP devices are used together, the element table is read from drum only once, thus saving up to 4700 words of memory by not duplicating the element table.

THE FORCETYPE PROCEDURE

FORCETYPE may be used to change the type of an element being

written in the program complex file by the PCF or SLIP device. The procedure call has the form

FORCETYPE(<integer exp>)

The type of the element currently being written in the complex will be set equal to the value of <integer exp>.

To be effective FORCETYPE should be called after the first card of the element has been written, and before the entry of this element has been completed. Therefore FORCETYPE should be called before the first card of the next element having different name/version is started, or, if the current element is the last element to be entered into the complex, it should be called before the termination of the program.

EDIT

The I/O device EDIT has been added to the list of devices which may be used from READ and WRITE. The call is

READ(EDIT(<string>),<parameter list>

WRITE(EDIT(<string>),<parameter list>

<parameter list> has the same form as in a call on READ or WRITE using CARDS or PRINTER, respectively.

In the first case the effect is to use the <string> in place of the card reader. Every time an activation phrase is encountered by the READ routine, EDIT will pass 80 characters from <string>. If more than one activation phase is present in the format, EDIT will pass sequential sets of 80 characters. If an attempt is made to pass more sets of 80 characters than are present in <string>, the READ will terminate, with action taken as follows:

If the <EOF label> is present in the I/O list, then
control will be transferred to it.
If not, the program is aborted, and the message for
insufficient data for program is printed.

All line skips are ignored. The implied format when reading is the same as for the card reader.

In the case of WRITE(EDIT(<string>),...) the effect is to use the <string> in place of the 1004 printer. The <string> is blank filled upon entry to EDIT. Every time an activation phrase is encountered by the WRITE routine, EDIT is passed 132 characters from WRITE. Sufficient

characters are skipped to satisfy the line spacing, and then the characters are inserted in the <string>. Eject format phrases are ignored. AØ.d activation phrases have the same effect as A1.d.

If an attempt is made to pass more sets of 132 characters than can be fit into <string> the WRITE operation will be terminated with action as follows:

If the <EOI label> is present in the I/O list, then control will be transferred to it.

If not, the EDITing is aborted, the message "string too short for EDIT" is printed, and control is transferred to the Algol error routine.

EXAMPLES OF USAGE:

```
(1)  INTEGER I,J,L $ REAL X,Y $ STRING B(10) $
     STRING A(80) $ FORMAT F1(A,S80), F2(A,X1, 3I4, 2F15.8), F3(A,S20,3I10,2F15.8) $
     READ(F1,A) $
     IF A(1) EQL '1' THEN READ(EDIT(A),F2,I,J,K,X,Y)
     ELSE READ(EDIT(A),F3,B,I,J,K,X,Y)
(2)  FORMAT F(S2,D15.8,A1) $ STRING A(132) $ REAL X $
     WRITE(EDIT(A),F,'X=', X) $
     CHAR(0.0,0.2,A(1,20))
```

CORE

CORE is designed to be a high speed random access device to be used in place of drum or drum tapes when the amount of data is relatively small but not easily stored in arrays because of unknown amounts or types. CORE also provides for retrieval of data by key where the number of keys is large or vary in type. CORE maintains a dictionary or directory of the information by their keys so that the information can be found quickly. As implied by the name the data is kept in core memory.

There are three uses of the CORE device:

```
WRITE(<core device>,<parameter list>)
READ(<core device>,<parameter list>)
RELEASE(<core device list>)
```

<parameter list> is any list of data and labels permitted in READ and WRITE for DRUM. The <core device> has three forms:

```
CORE(<key expression>)
CORE
CORE(<key expression>,<block size>)
```

The <key expression> is any simple variable of type INTEGER, REAL, BOOLEAN, COMPLEX, REAL2, or STRING. The value and type of the <key expression> is the key associated with the information. This means that 1, 1.0, TRUE, <1.0,0.0>, and '1' are unique keys. The second use of CORE designates a special key called null for which no look up is necessary in the dictionary. This saves time when no key is necessary for the data.

The information written is kept in blocks of memory, each of which is initially ten words long. Each block also uses three extra words internally. The second parameter to core is a positive integer less than 2049. This changes the block size for all further writes until the block size is changed again. Partial blocks of memory are never returned so that the size should be adjusted to maximize use of core.

WRITE causes the data in the parameter list to be moved to available blocks of memory and the key expression to be associated with this information. If a previous WRITE with the same key exists then the information from the previous WRITE is lost and the memory used is either reused for the new WRITE or returned to available storage space by blocks. If no available memory exists for a block in a WRITE then control will be returned to the label in the <parameter list> if one exists; otherwise the run will be aborted. If computation is to continue, the information in the new WRITE should be released since it is incomplete.

READ causes the information associated with the key to be transferred to the variables in <parameter list>. The information is not changed by the READ and may be reread. Two labels may be used in the <parameter list> with READ. Control will return to the first label if more data is specified in the READ than has been written. The second label is used if no information had been written with this key or if the information and key had been wiped out by a RELEASE statement. If the appropriate label is not provided the run will be aborted when either of these error conditions occurs.

The procedure RELEASE allows information and keys no longer needed to be erased and the core memory space returned to the available space list so that it can be used by the rest of the program. The <core device list> is a list of core devices with keys specifying which are to be released from further use. The release of information previously released

or never written does not cause an error, but is simply ignored.

Note that if the information is too large to be kept in core then the key may just be the key to the drum address where the data is stored. However, it is up to the user to manage the reuse of drum space.

EXAMPLE:

```
      INTEGER I, L, MAXL, K, CNT $
      STRING WORD(30) $
      EXTERNAL PROCEDURE NEXT $
COMMENT NEXT IS A PROCEDURE WHICH SCANS CARDS AND PUTS THE NEXT IDENTIFIER
      IN WORD AND ITS LENGTH IN L $
      K = 0 $
LOOP..      NEXT(WORD,L) $
            IF L EQL 0 THEN GO TO OUTPUT $ COMMENT END OF DATA $
            READ(CORE(WORD),I,ERR,NEW) $
COMMENT FIND INDEX I FOR THE IDENTIFIER IN WORD.  IF NONE IS FOUND
      TRANSFER CONTROL TO NEW FOR A NEW ENTRY.  $
            READ(CORE(I),WORD,CNT+1) $ COMMENT UPDATE COUNT $
            GO TO LOOP $
NEW..       K=K+1 $ COMMENT INCREMENT INDEX $
            WRITE(CORE (K),1) $ COMMENT SET COUNT TO 1 $
            READ(CORE(-L),I,ERR,NEWCNT) $
COMMENT GET COUNT OF NUMBER OF DIFFERENT IDENTIFIERS OF LENGTH L OR GO
      TO NEWCNT IF THIS IS THE FIRST IDENTIFIER OF LENGTH L $
            WRITE(CORE(-L),I+1) $ COMMENT UPDATE COUNT $
            GO TO LOOP $
NEWCNT..    WRITE(CORE(-L),1) $ COMMENT ONE IDENTIFIER THIS LENGTH $
            MAXL=MAX(L,MAXL) $ COMMENT KEEP MAX LENGTH $
            GO TO LOOP $
OUTPUT..    FOR L=1 STEP 1 UNTIL MAXL DO
            BEGIN
                READ(CORE(-L),I,ERR,NONE) $
                WRITE(L,I) $
COMMENT WRITE NUMBER OF IDENTIFIERS OF LENGTH L $
                RELEASE(CORE(-L))
NONE..       END $
```

```
            FOR L=1 STEP 1 UNTIL K DO
            BEGIN
                READ(CORE(L),WORD,CNT) $
COMMENT READ EACH IDENTIFIER AND THE NUMBER OF ITS OCCURRENCES $
                WRITE(WORD,CNT) $
                RELEASE(CORE(L),CORE(WORD))
             END $
             GO TO EXIT $
ERR..        WRITE('/-/-/-/ C O R E  F A U L T /-/-/-/') $
EXIT..
```

APPENDIX VII

FALTRAN - TRANSLATION FROM FORTRAN TO ALGOL

The processor Faltran translates a Fortran source program into an Algol source program. It allows the user to do any or all or the following operations simultaneously (i.e., with a single call on Faltran):

1) translate a Fortran mainline program with or without internal subprograms;
2) use correction cards to modify the Fortran program;
3) enter the updated symbolic Fortran program in the user's program complex file;
4) enter the resulting Algol symbolic in the user's PCF;
5) include externally defined Fortran subprograms (after they have been translated) in the resulting symbolic Algol program.

Fortran statements that present problems for translation into equivalent Algol symbolic are marked by the Faltran translater with error messages in the printed listing. These statements are discussed later in this appendix (see LIMITATIONS OF THE TRANSLATION PROCESS).

THE FALTRAN PROCESSOR CARD

The processor call card for Faltran when the input is from cards has the form

$$\frac{7}{8}\langle\text{options}\rangle\Delta\text{FAL}\Delta\langle\text{element}_1\rangle,\langle\text{element}_2\rangle,\langle\text{element}_3\rangle$$

and for input from drum the form is

$$\frac{7}{8}\langle\text{options}\rangle\Delta\text{FAL},*\Delta\langle\text{element}_1\rangle,\langle\text{element}_2\rangle,\langle\text{element}_3\rangle$$

where $\langle\text{element}_1\rangle$ has the form

$$\langle\text{name}_i\rangle/\langle\text{version}_i\rangle(\langle\text{cycle}_i\rangle)$$

and the options are as follows:

OPTION LETTER	DESCRIPTION
N	Do not list the input Fortran or the translated Algol
I	Do not list the input Fortran
O	Do not list the translated Algol
X	Abort the processor if an error is found

<element$_1$> is the name/version of the input Fortran element, <element$_2$> is the name/version of the updated Fortran element that is entered into the complex, and <element$_3$> is the name/version of the resulting Algol source program, which is also entered into the complex.

If it is desired to translate several subprograms and include them with a mainline program then an EOF card must follow the mainline program and this must be followed by a card of the form

<element$_4$>,<element$_5$>

Here <element$_1$> has the same form as above, <element$_4$> is the name/version of the input Fortran subprogram and <element$_5$> is the name/version of the updated Fortran program, which is entered into the program complex.

If the main program, <element$_1$>, is input from drum then <element$_4$> must also be on the drum. <element$_5$> may be omitted if the updated Fortran subprogram is not to be saved on drum. Correction cards may be used on <element$_4$>.

It is assumed that normally these subprograms are compiled separately from the main program. After the last subprogram (or the main program if there are no separately compiled subprograms included) there must be a control card - any control card except an EOF card. Of course, it is possible to include subprograms with the main program to be translated by Faltran, just as it is possible to include internal subprograms in a main program to be compiled by Fortran.

LIMITATIONS OF THE TRANSLATION PROCESS

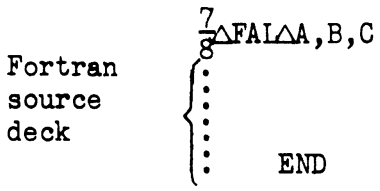
Some Fortran statements are not translated at all by Faltran. Therefore, Faltran output should be scrutinized before attempting to run it through the Algol compiler.

The main problem is FORMAT statements, which are never processed by Faltran. The programmer must translate these according to the needs of his program.

Other features of Fortran that must be handled by the user as he sees fit include Hollerith strings, DATA statements and BLOCK DATA programs. These present problems, the resolution of which is either impossible or incompatible with the goal of an efficient translator.

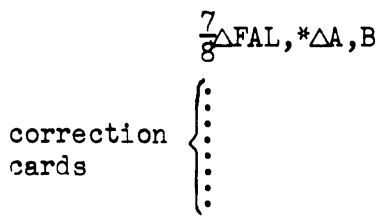
EXAMPLES

Example 1:



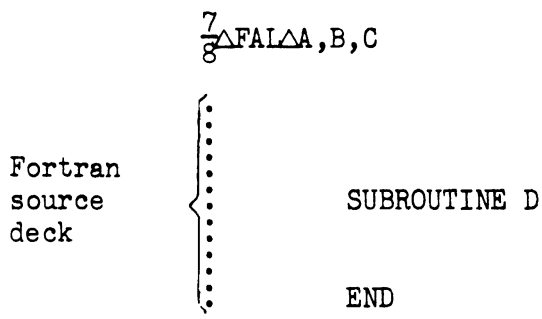
The updated Fortran symbolic is named B, the resulting Algol is named C, and both B and C are entered into the user's PCF.

Example 2:



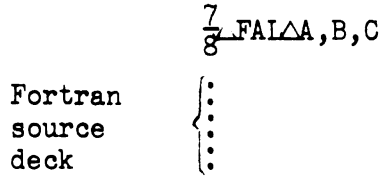
The Fortran program named A is read from the user's PCF, the corrected version of A is entered into the PCF and named B, and the translated Algol is entered under the name C.

Example 3:



The mainline A and an internal subroutine D are translated into a single Algol program named C.

Example 4:



```
          7
          8△EOF

          D,E
correction :
cards      :
```

The Fortran mainline A and the external subroutine D are together translated into the Algol symbolic named C. The updated version of D is entered into the complex under the name E. The symbolic for D must be in the user's PCF on drum when the Faltran is called.

APPENDIX VIII

MACHINE- AND SYSTEM-DEPENDENT INTRINSIC FUNCTIONS AND PROCEDURES

This appendix is devoted to intrinsic functions that are specific to the hardware of the Univac 1107, the Exec III operating system and the structure of the Algol compiler and library. The interested reader is referred to the detailed documentation of each for relevant details. These functions were originally created to facilitate the writing of processors (e.g., compilers) in Algol.

FIELD (<exp$_1$>,<exp$_2$>,<exp$_3$>)

The function may be written on either the left-hand or right-hand side of an assignment statement. On the right-hand side it allows the extraction of a specified bit field of an 1107 36-bit word. When written on the left it allows replacement of a bit field.

In either case the word is designated by <exp$_1$>, which may be of any type. <exp$_1$> will not be converted unless it is a STRING expression, in which case the first six characters will be used. If the characters in the string number is less than six they will be right-justified and master-space filled before the FIELD operation takes place.

The expression <exp$_2$> denotes the starting bit of the field (bits are numbered 1 to 36, left to right, across the word) and <exp$_3$> denotes the number of bits in the field. Both <exp$_2$> and <exp$_3$> may be of type INTEGER, REAL, REAL2, or STRING and will be converted to integers if necessary. If <exp$_3$> is omitted from the call, the length is taken to be one bit and if both <exp$_2$> and <exp$_3$> are omitted the starting position will be bit 1 and the length 36 bits (i.e., the entire 1107 word at the address specified by <exp$_1$> is used.)

When FIELD is used on the left of an assignment statement only the part of the word determined by <exp$_2$> and <exp$_3$> is affected and if <exp$_1$> denotes a STRING a maximum of six characters may be replaced.

Multiple replacement statements involving FIELD are undefined (e.g., FIELD(F,1,5) = FIELD(6,32) = FIELD(S(I)).

The FIELD function is undefined for these values of the parameters:

$\langle exp_2 \rangle = 0$; $\langle exp_3 \rangle = 0$; $\langle exp_2 \rangle > 36$; $\langle exp_3 \rangle > 37 - \langle exp_2 \rangle$.

The values of $\langle exp_2 \rangle$ and $\langle exp_3 \rangle$ are not checked at execution time.

FIELDS ($\langle exp_1 \rangle$,$\langle exp_2 \rangle$,$\langle exp_3 \rangle$)

The arguments of FIELDS have the same meaning as those of the FIELD function except that if $\langle exp_1 \rangle$ is of type STRING only, the FIELDS function acts upon the string descriptor rather than the characters in the string.

DSC ($\langle exp_1 \rangle$,$\langle exp_2 \rangle$)	"Double Shift Circular"
DSA ($\langle exp_1 \rangle$,$\langle exp_2 \rangle$)	"Double Shift Arithmetic"
DSL ($\langle exp_1 \rangle$,$\langle exp_2 \rangle$)	"Double Shift Logical"
SSC ($\langle exp_1 \rangle$,$\langle exp_2 \rangle$)	"Single Shift Circular"
SSA ($\langle exp_1 \rangle$,$\langle exp_2 \rangle$)	"Single Shift Arithmetic"
SSL ($\langle exp_1 \rangle$,$\langle exp_2 \rangle$)	"Single Shift Logical"

Each of these functions is based on the Sleuth instruction whose mnemonic is the same as the function name. The variable specified by $\langle exp_1 \rangle$ is loaded into register A2 (A2,A3 for two word variables) and a shift of $\langle exp_2 \rangle$ places is performed on A2. Then the contents of A2 (A2,A3) is stored according to the Algol statement.

MEMORY (<exp>)	"core memory"
PARTBL (<exp>)	"parameter table"
MCR (<exp>)	"monitor communications region"

These functions may be used to store into or load from any word of core memory that Exec III does not regard as sacred.

The MEMORY function retrieves or stores into the word of core denoted by <exp>. The PARTBL function serves the same purpose except that the starting address of the Exec parameter table is added to <exp> to form an effective address. The MCR function is similar to the PARTBL function except that the starting address of the Exec Monitor Communications Region is added to <exp> to form an effective address.

The reader is referred to the most recent Exec documentation for the addresses of core that may be accessed without interference by the Exec and the significance of those locations.

EXIT

EXITNORMAL

EXITERROR

EXITABORT

Any of these four intrinsic procedures may be called at any point in an Algol program to cause a jump to an Exec entry point. None of them requires a parameter.

The functions EXIT and EXITNORMAL are identical in their effect, which is to cause a

```
J       MEXIT$
```

to be executed.

EXITERROR causes a

```
J       MERR$
```

to be executed.

EXITABORT causes a

```
J       MXXX$
```

to be executed.

If the Algol J-option is in effect then the jumps will be made to the corresponding processor exits. It should be noted that the usual program termination action provided by the Algol library is circumvented when these special exit functions are invoked.

PARITYEVEN(<arith exp>)

PARITYODD(<arith exp>)

These functions have BOOLEAN results and before they are evaluated the parameter is converted to type INTEGER if it is not already of that type.

The result of PARITYEVEN is TRUE if the 1107 internal representation of the parameter has an even number of bits, and FALSE otherwise. The function PARITYODD has a value of TRUE in case of odd parity, FALSE otherwise.

APPENDIX IX

A CULL FOR USE WITH ALGOL PROGRAMS

The processor called Alcull produces a sorted list of the occurrences, by block number and card number, of all identifiers defined in an Algol program, as well as all reserved words used in the program. A listing of the Algol program may also be printed if desired.

The card number on which an identifier is defined is denoted by an asterisk and appears first in the list for that identifier, even though, in the case of a label, it may not be the first card on which the identifier is used.

An example of the output is the following:

```
ANYIO          BOOLEAN
          1:    170*   1940   2767   2769
ANYOUT         OWN BOOLEAN
          4:    303*    307    334    341    366    374
          5:    322
ARRAY          RESERVED WORD
          1:    175     178
```

The BOOLEAN variable ANYIO was defined in block 1 on card 170 and referenced on cards 1940, 2767 and 2769. The OWN BOOLEAN variable ANYOUT was defined in block 4 on card 303 and referenced on card 322 in block 5, among others. The reserved word ARRAY was used in block 1 on cards 175 and 178.

The processor card for Alcull has the form

$$\frac{7}{8}\langle\text{options}\rangle\Delta\text{ALC}$$

if the input is from cards. If the input is from the drum then the processor card is

$$\frac{7}{8}\langle\text{options}\rangle\Delta\text{ALC},*\Delta\langle\text{element name-version-cycle}\rangle.$$

The relevant options for Alcull are

F - scan all 80 columns

N - do not list the input program

Best results are obtained when culling syntactically correct Algol programs.

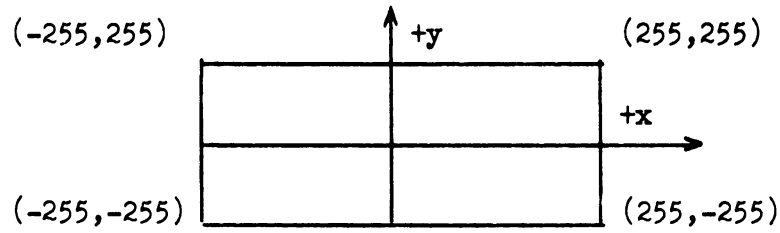
APPENDIX X

CATHODE RAY TUBE (CRT) PROCEDURES FOR ALGOL PROGRAMS

Procedures are available to display pictures on the CRT. These procedures must be CURred in from tape when needed.

Coordinates

The CRT display area is a square with the following coordinate system:



THE DISPLAY PROCEDURE

The DISPLAY procedure has the five forms:

FIRST FORM: DISPLAY(<time>,<display items>)

Here <time> is the time, in seconds, that the picture is to stay on the screen. It is an INTEGER, REAL, or REAL2 expression, and must be within the limits 1/60 sec LEQ <time> LEQ 1 hour.

<display items> is a series of <item>'s which describe the picture. Each <item> is a series of two, three or four parameters as described below:

(1) The group of parameters

<string exp>,<coordinates>

will display the string with the leftmost character at the coordinates given. Only that part of the string which is in the display area will be displayed.

<coordinates> are either (a) two INTEGER, REAL or REAL2 expressions giving the abscissa and ordinate, respectively, or (b) a single COMPLEX expression whose real and imaginary parts give the abscissa and ordinate, respectively.

(2) The group

<coordinates$_1$>,<coordinates$_2$>

will display a straight line segment from <coordinates$_1$> to <coordinates$_2$>.

Only that part of the segment which is in the display area will be displayed.

Display items may be generated by simple or generative lists, or by calls on special procedures to be described later.

The DISPLAY procedure, when called by the first form, will edit the time and items into a 500-word core buffer, start the picture, and return to the main program.

SECOND FORM: DISPLAY (<integer array>,<display items>)
where <display items> are as before and <integer array> is one-dimensional. The DISPLAY routine will edit the items into CRT output data and put them into <integer array>. If <integer array> already contains such data, the new data will be added at the end. The first element of <integer array> (lowest subscript) contains the data count (so it should be initialized to zero), and subsequent elements contain the data. No picture is displayed.

THIRD FORM: DISPLAY (<time>,<integer array>)
will display the data edited into <integer array> by the second form.

FOURTH FORM: DISPLAY (<time>)
This is essentially the first form with no <display items>. It will display a blank screen for the time given.

FIFTH FORM: DISPLAY
This should be programmed at the end of the Algol program. Nothing is displayed, but the routine waits until the previous picture is done before continuing.

Declaring external procedures

The identifier DISPLAY is recognized by the compiler without being defined. However, all the procedures described below must be defined by the following declarations:

```
EXTERNAL PROCEDURE DABORT, DEOF, DINPUT, DERR $
EXTERNAL INTEGER PROCEDURE BROKEN, SIML $
EXTERNAL SLEUTH INTEGER PROCEDURE ARC, LETTER $
```

ABORTING A PICTURE

FROM THE CONSOLE. To abort a picture from the console, turn selective

jump #4 on and then off. The picture will vanish and the next picture (if any) will begin.

FROM THE PROGRAM. To abort a picture from the program, simply execute the procedure call:

DABORT (no parameters)

ABORTING A SERIES OF PICTURES

To abort a series of pictures, put the procedure call

DEOF (<label>)

at the appropriate place in or after the series. Then while the series is visible on the CRT, turn selective jump #5 on. This aborts every picture until the above procedure call is encountered. Then control hangs in a tight loop until selective jump #5 is turned off. Then transfer is made to <label> or, if <label> is omitted, to the next statement.

EXAMPLE:

```
COMMENT ROTATING LINE $
INTEGER I $ REAL A $
EXTERNAL PROCEDURE DEOF $
LOCAL LABEL L $
A = 0 $
FOR I = 1 STEP 1 UNTIL 15*60 DO BEGIN
   DISPLAY (1.15,-100*COS(A),-100*SIN(A),100*COS(A),100*SIN(A)) $
   A = A + 0.05 $
   DEOF(L) END $
L..
```

BROKEN LINES

If the procedure call

BROKEN (<coordinates$_1$>,<coordinates$_2$>, ...,<coordinates$_N$>)

is passed as a display item to DISPLAY, it will generate the proper items to draw a broken (polygonal) line from <coordinates$_1$> to <coordinates$_2$> to ... to <coordinates$_N$>. Coordinates may be generated by simple or generative lists. The BROKEN procedure, therefore, shortens notation by eliminating the need for repeating coordinates of interior points.

EXAMPLE:

```
COMMENT A HUGE LETTER "S" $
DISPLAY ( 60.0, BROKEN ( 50,100, -50,100, -50,0, 50,0,
                         50,-100, -50,-100 ) )
```

ARCS

The procedure call

ARC (<coordinates>,<radius>,<angle$_1$>,<angle$_2$>,<integer exp>)

generates an approximation to an arc of radius <radius> about the point <coordinates> from <angle$_1$> to <angle$_2$>, using <integer exp> chords. Here <radius> is an INTEGER, REAL or REAL2 expression and <angle$_1$> and <angle$_2$> are INTEGER, REAL or REAL2 expressions in units of degrees.

BLOCK LETTERING

The procedure call

LETTER (<coordinates>,<Δx>,<Δy>,<kind>,<string exp>)

will generate the appropriate line segments to display the <string exp> in block letters. Here <coordinates are the coordinates of the lower left corner of the first letter, and <Δx> and <Δy> are two INTEGER, REAL or REAL2 expressions giving the components of a vector whose length ($\sqrt{\Delta x^2+\Delta y^2}$) is the height of the lettering and whose direction in the direction of the lettering. If <Δy> is omitted, it is presumed to be zero, giving horizontal lettering of height <Δx>. <kind> is a three-digit integer which determines the kind of lettering used, according to the following scheme:

hundreds	tens	units
0 = Roman	0 = capital	0 = regular
1 = Greek	1 = small	1 = superscript
	2 = slant capital	2 = subscript
	3 = slant small	

At present the only <kind>'s which work are 010 and 000, or Roman small regular and Roman capital regular, respectively.

The <string expression> is the string to be edited to block letters. All capital characters except a few will be displayed in block form which is much like the printed form. Small characters other than letters will, for the most part, be the same as capitals, but the spacing will be closer. However, a dollar sign in a Roman small regular (<kind>=10) string will be skipped and the next character will be made capital. The character after that will be again small.

The parameters may end with the first string, or additional groups of parameters of the form

<changes>,<string exp>

may be added. Here <string exp> is a continuation of the previous string and <changes> are the changes in the editing process. These <changes> have three forms:

FIRST FORM: <coordinates>

Here <coordinates> are two INTEGER, REAL or REAL2 variables or a single COMPLE variable (not an expression or constant). The coordinates of the lower left-hand corner of the next character are stored into these variables. Then editing continues as before.

SECOND FORM: <kind>

Here <kind> is the new kind under which the new string is to be edited.

THIRD FORM: <coordinates>,<kind>

This combines the effects of the first two forms.

EXAMPLE: The procedure call

```
DISPLAY ( 60, LETTER ( -200.0,0,20,20,10, '$THE', X1,Y1,30,
                  'Δ $TITANIC', X2,Y2 ), X1+10,Y1-10, X2+10,Y2-10 )
```

will display the picture:

THE TITANIC

ERROR MESSAGES

If the DISPLAY procedure cannot partition its arguments properly into items, the error message IMPROPER NUMBER OF PARAMETERS TO DISPLAY will result. If the core buffer capacity is exceeded, the message TOO MANY ITEMS IN DISPLAY will result. Other messages are self-explanatory.

If a run-time error occurs while a picture is in process, the error message will be lost and the registers will be dumped twice. To prevent this, the procedure call

DERR (no parameters)

should be made at the beginning of a program being debugged. This procedure will overstore the error routine with instructions which cause it to wait until the current picture is finished. Then the error message and line numbe

will be printed out normally.

INTERROGATION OF CRT SWITCHES FROM ALGOL

The switches on the CRT may be interrogated in al Algol program by using the library function named CRTSW. The result of CRTSW is of type BOOLEAN.

The call on CRTSW has the form

CRTSW (<integer exp>)

where the value of <integer exp> specifies the number of the switch to be interrogated (the switches are numbered from one to eighteen). If the indicated switch is UP and the black button is pressed, the value of CRTSW is TRUE, otherwise the value is FALSE.

APPENDIX XI

CHARACTER DEFINITIONS FOR THE 1107

Octal Code	Symbol	Card Code	1004	HSP 600 lpm	Console Printer	1090 CRT
00	M.S.	7-8	@	Space	M.S.	△
01	U.C.	12-5-8	[	Space	U.C.	⌐
02	L.C.	11-5-8	]	Space	L.C.	¬
03	L.F.	12-7-8	#	Space	L.F.	|
04	C.R.	11-7-8	△	Space	C.R.	___
05	Space	Blank	Space	Space	Space	Space
06	A	12-1	A	A	A	A
07	B	12-2	B	B	B	B
10	C	12-3	C	C	C	C
11	D	12-4	D	D	D	D
12	E	12-5	E	E	E	E
13	F	12-6	F	F	F	F
14	G	12-7	G	G	G	G
15	H	12-8	H	H	H	H
16	I	12-9	I	I	I	I
17	J	11-1	J	J	J	J
20	K	11-2	K	K	K	K
21	L	11-3	L	L	L	L
22	M	11-4	M	M	M	M
23	N	11-5	N	N	N	N
24	O	11-6	O	O	O	O
25	P	11-7	P	P	P	P
26	Q	11-8	Q	Q	Q	Q
27	R	11-9	R	R	R	R
30	S	0-2	S	S	S	S
31	T	0-3	T	T	T	T
32	U	0-4	U	U	U	U
33	V	0-5	V	V	V	V
34	W	0-6	W	W	W	W
35	X	0-7	X	X	X	X
36	Y	0-8	Y	Y	Y	Y
37	Z	0-9	Z	Z	Z	Z

NOTE: The offline 1004 follows the above table. However, under Exec III, when the 1004 is on line, the characters "@" and "△" are interchanged.

Octal Code	Symbol	Card Code	1004	HSP 600 lpm	Console Printer	1090 CRT
40	)	12-4-8	)	)	)	)
41	-	11	-	-	-	-
42	+	12	+	+	+	+
43	<	12-6-8	<	<	<	<
44	=	3-8	=	=	=	=
45	>	6-8	>	>	>	>
46	¬	2-8	&	&	¬	&
47	$	11-3-8	$	$	$	$
50	*	11-4-8	*	*	*	*
51	(	0-4-8	(	(	(	(
52	"	0-5-8	%	Space	"	"
53	:	5-8	:	:	:	:
54	?	12-0	?	Space	?	?
55	!	11-0	!	Space	!	!
56	,	0-3-8	,	,	,	,
57	Stop	0-6-8	\	Space	⟨S⟩	%
60	0	0	0	0	0	0
61	1	1	1	1	1	1
62	2	2	2	2	2	2
63	3	3	3	3	3	3
64	4	4	4	4	4	4
65	5	5	5	5	5	5
66	6	6	6	6	6	6
67	7	7	7	7	7	7
70	8	8	8	8	8	8
71	9	9	9	9	9	9
72	'	4-8	'	'	'	'
73	;	11-6-8	;	Space	;	;
74	/	0-1	/	/	/	/
75	.	12-3-8	.	.	.	.
76	Special	0-7-8	□	Space	□	□
77	Idle	0-2-8	≠	Stop (N.P.)	↑	↑

I N D E X