BURROUGHS

A REPRESENTATION OF ALGOL FOR USE WITH

ALGEBRAIC

THE BURROUGHS 220 DATA-PROCESSING SYSTEM

COMPILER

BULLETIN 220-21011-D
January 1961

a reference manual

BURROUGHS ALGEBRAIC COMPILER

Sales Technical Services
EQUIPMENT & SYSTEMS
MARKETING DIVISION
BURROUGHS CORPORATION
Detroit 32, Michigan

SECOND PRINTING

MONOTYPE COMPOSITION  PRINTED IN USA

table of contents

Table of Contents (continued)

SYSTEM REQUIREMENTS

ORGANIZATION OF THE MANUAL

ADDITIONAL COPIES

I . . .

introduction

THIS BOOK IS INTENDED as a reference manual in the use of the BURROUGHS ALGEBRAIC COMPILER. The BURROUGHS ALGEBRAIC COMPILER is a hardware representation of ALGOL; it accepts symbolic programs and produces machine-language programs for the BURROUGHS 220 Electronic Data Processing System. A full description of the evolution and present status of ALGOL is available elsewhere.†

This compiler utilizes a BURROUGHS 220 consisting of at least the following components: 5,000 words of core storage, CARDATRON,® with one input and one output (LINE PRINTER) station, and two magnetic-tape storage units.

The compiler itself consists of approximately 3,500 instructions and 2,000 words of stored tables.

The compiler reads the symbolic program from punched cards and prints out a copy on the LINE PRINTER. This listing includes a count of the cells used by the compiled program, together with any diagnostic messages to the programer. The machine-language representation of the compiled program is written on magnetic tape at approximately 500 instructions per minute. When all the symbolic cards have been processed, routines are copied from the compiler library onto the output tape, along with an appropriate loading routine. At the programer's option, the output tape may also include diagnostic routines tailored to the particular program which has been compiled. Provision is made for the inclusion in the compiled program of machine-language routines. At the end of compilation, the compiled program may be loaded and executed immediately.

The text of this reference manual consists principally of definitions and rules for the use of the compiler, 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, of the contents of the library, etc.

Whenever a term is defined, it is italicized in the defining sentence. References in the index to definitions are also italicized. Script letters are used in the text to denote generic representations; for example, $\mathcal{E}$ is used to represent an expression and $\mathcal{S}$ to represent a statement. (In APPENDIX D, in accordance with ALGOL representation, some Greek letters have been employed.) 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.

BURROUGHS CORPORATION is pleased to work closely with the Subcommittee on ALGOL of CUE (Coöperating Users' Exchange) in the task of maintaining this and other literature concerning the BURROUGHS ALGEBRAIC COMPILER.

Although every effort has been made to publish this manual without errors, success in such an endeavor is seldom completely attained. Constructive criticisms of the contents of this manual will be appreciated by the authors and publishers. Please address them to:

MANAGER, AUTOMATIC PROGRAMING
Burroughs Corporation
460 Sierra Madre Villa
Pasadena, California, USA

Additional copies of this manual may be obtained from your BURROUGHS CORPORATION representative.

† See *Communications of the ACM*, vol. 1, no. 12, pp. 8-22; and vol. 3, no. 5, pp. 299-313.

CHARACTER SETS
IDENTIFIERS
QUANTITIES
VARIABLES
CONSTANTS

II . . .

elements of the compiler language

CHARACTERS

THE BURROUGHS ALGEBRAIC COMPILER employs a character set which is commonly available as a variant of the usual Hollerith code.† These characters are:

Scientific Character set	Hollerith Equivalents
THE ROMAN ALPHABET	
A, B, ..., Z	A, B, ..., Z
THE ARABIC NUMERALS	
0, 1, ..., 9	0, 1, ..., 9
SPECIAL CHARACTERS	
+	&
−	− or @
=	#
(	%
)	□
.	.
,	,
;	$
/	/
*	*
⟨space⟩	⟨space⟩

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

..	:
...	ellipsis
**	base-10 scale factor appended

† All of the examples in this manual are printed with the scientific character set. If card equipment with FORTRAN characters is used, these same characters will be printed, with the exception of the semicolon (;), which will print as a dollar sign ($). (The type wheel is variation 'F'.) Card equipment with 'standard' characters will print in Hollerith equivalents.

‡ See page 10-3 for restriction on the length of identifiers to be monitored.

From these characters statements are constructed which are translated by the compiler into the form suitable for execution by the BURROUGHS 220.

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
~	is equivalent to / has the form of
~~	ellipsis
⟨ ⟩	brackets
ρ	relational operator
o	arithmetic operator
|	or
#	space

IDENTIFIERS

The fundamental construct of the compiler language is the identifier. Identifiers are used to name the various things which make up a program, for example, variables, functions, labels, subroutines, etc. An identifier is composed of a string of letters, or letters and digits, not exceeding 50 characters in length.‡ The first character of an identifier must be a letter; no special characters (including spaces) may be embedded within an identifier.

In addition, a few identifiers are reserved for special use as operators, punctuation marks, and as names of library functions.

These reserved identifiers may not be used by the programer in any context other than that set down in this manual. A list of the reserved identifiers is given in APPENDIX B. Any other identifiers may be used at will.

EXAMPLES:

```
Z
GAMMA
SN2N1
SIERRAMADREVILLAAVENUE
A374
YOUNGLADYOFCHICHESTER540
RUNGEKUTTAGILL
```

QUANTITIES

The BURROUGHS Algebraic Compiler is concerned with the manipulation of three types of quantities: *Floating-point quantities*, *integer quantities*, and *Boolean quantities*.

Floating-point quantities are numbers which may have both an integral and a fractional part. They are used to represent the class of real numbers to an accuracy of eight significant digits, the maximum permitted by the word length of the BURROUGHS 220. The magnitude of a floating-point quantity must be less than 10^{50}. Any floating-point quantity which is smaller in magnitude than 0.1×10^{-50} is represented by zero.

Integer quantities are those numbers which do not have a fractional part, and which represent the class of integers which can be expressed in the word length of the BURROUGHS 220, i.e., integer quantities must be smaller than 10^{10} in magnitude.

Boolean quantities represent truth values. The only values for Boolean quantities are one, meaning ***true***, and zero, meaning ***false***.

A program may contain quantities of any or all of these three types. The programer assigns the types of the variables, evaluated functions, and expressions which appear in his program. (See CHAPTER V.) The type of a constant depends upon context and form.

VARIABLES

Variables treated by this compiler are of two kinds—simple variables and variables with subscripts. A *simple variable* represents a single quantity and is denoted by an identifier; *a variable with subscripts* represents a single element of an array and is denoted by the identifier which names the array, followed by a subscript list enclosed in parentheses. The 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)
Z(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. Even floating-point values are allowed, in which case the floating-point number is truncated—the fractional part dropped—to an integer. Each subscript must have a value which is not less than unity and not greater than the maximum specified for that array by the ARRAY declaration (see CHAPTER V). The number of subscript expressions must equal the number of dimensions of the array.

Whether a variable represents a floating-point, integer, or Boolean quantity is determined by the 'declaration of type' described in CHAPTER V.

CONSTANTS

Integer Constants

Integer constants are represented by a string of digits. A maximum of ten significant digits is allowed. Leading zeros are ignored. Spaces may not be imbedded within an integer.

EXAMPLES:

```
    0
   17
16384
 2111
```

Floating-Point Constants

Floating-point constants are represented by a string of digits which contains '.'—a decimal point. *The decimal point may not appear at the beginning or end of the string; it must be imbedded within it.* A floating-point constant may contain a maximum of eight significant digits. Leading zeros are not counted toward this maximum.

EXAMPLES:

```
 3.1415927
43.0
 0.00006174205
```

If desired, a scale factor may be appended to a floating-point constant to indicate that it is to be multiplied by the indicated power of 10. This scale factor is written as two asterisks 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. The magnitude of such a floating-point number must not exceed $0.99999999 \times 10^{49}$.

EXAMPLES:

2.6**5 means 2.6×10^5 or 260,000

1.7**−3 means 1.7×10^{-3} or 0.0017

A third option allows a floating-point 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 not be used to specify a floating-point number—it is not valid to use **−2 to indicate 10^{-2}; it must be written 1**−2 or 1.0**−2, etc.

Boolean Constants

Only two *Boolean constants* are allowed: Zero (written as 0) means ***false***, and one (written as 1) means ***true.***

EVALUATED FUNCTIONS

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 VIII contains a complete description of the use of functions and the manner in which they are defined.) For the moment we will assume that a function acts on one or more quantities called *arguments* and produces a single quantity as a result. This resulting quantity is called an *evaluated function.*

GENERAL FORM:

$g(\varepsilon_1, \ldots, \varepsilon_q)$

where g is an identifier which names the function and ε_1 through ε_q are expressions which are the arguments of the function.

EXAMPLES:

```
SIN(X)
SQRT(B**2-4.A.C)
HYPERGEOM(A,B,C,Z)
LOG(SIN(THETA-ALPHA/2))
PEIRCE(P,Q)
```

The type of a function depends on the manner in which the function was defined. The type required for each of the arguments is also determined by the definition of the function. *It is the programer's responsibility to ensure that each of the arguments is of the proper type.*

ARITHMETIC EXPRESSIONS

BOOLEAN EXPRESSIONS

ARITHMETIC RELATIONS

III . . .

expressions

THE COMPILER deals with two kinds of expressions: Arithmetic expressions (those having numerical values) and Boolean expressions (those having truth 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, that is, B*2 has the meaning B^2. In addition to these five symbols, parentheses are employed to indicate that a specific order of evaluation is to be followed rather than the conventional order of evaluation. To be explicit, it is assumed—in the absence of parentheses to indicate otherwise—that exponentiation is performed before multiplication, multiplication before division, and division before addition and before subtraction. Convention in writing algebraic expressions suggests this ordering rather than that of assigning equal precedence to multiplication (·)† and division (/). As is customary in mathematical literature, the expressions A/B/C and A*B*C are regarded as ambiguous. Parentheses should be used to express the exact meaning desired.

A variable, a constant, or an evaluated function of floating or integer type will in itself constitute an arithmetic expression. Furthermore, if ε_1 and ε_2 are any arithmetic expressions and ε_3 is an arithmetic expression the first character of which is not a + or −, then each of the following combinations is also an arithmetic expression:

$\varepsilon_1 \cdot \varepsilon_2$	$\varepsilon_1 + \varepsilon_3$
$\varepsilon_1 / \varepsilon_2$	$\varepsilon_1 - \varepsilon_3$
$\varepsilon_1 * \varepsilon_2$	$+\varepsilon_3$
(ε_1)	$-\varepsilon_3$

† Represented on card equipment as a decimal point (.).

If ε_1 and ε_2 are both constants, the programer must write $(\varepsilon_1) \cdot (\varepsilon_2)$, to avoid conflict with the notation for constants.

EXAMPLES:

```
X + Y*2
C.SIN(N.3.1415927.F)
ARCTAN(HORIZ/VERTL) − ALPHA
(−B + SQRT(B*2 − 4.A.C))/2.A
(Z13*−3 + Z14*−3)/−17.2
A(I + 1, J + 1) − A(I + 1,1)/V(J)
```

Omission of the Multiplication Sign

In certain instances the '·' representing multiplication may be omitted. In general, this omission is possible wherever the lack of a '·' will not result in ambiguity.

More specifically, suppose that:

$\mathfrak{g}$ is an identifier specifying a simple variable, an array, or a function;
$\mathfrak{V}$ is an identifier specifying a simple variable;
$\mathfrak{N}$ is any constant; and
$\sim$ is the symbol for *is equivalent to*; then—

$)\mathfrak{g} \sim) \cdot \mathfrak{g}$
$\mathfrak{V}(\sim \mathfrak{V} \cdot ($
$)\mathfrak{N} \sim) \cdot \mathfrak{N}$
$\mathfrak{N}(\sim \mathfrak{N} \cdot ($
$) (\sim) \cdot ($
$\mathfrak{N}\mathfrak{g} \sim \mathfrak{N} \cdot \mathfrak{g}$

EXAMPLES:

```
4A.C
3(A + B)(A − B)
TAN(2X)ALPHA
2SIN(X)COS(X)
```

The Type of an Arithmetic Expression

The type of an arithmetic expression is determined by the types of its constituents. Suppose that $\mathcal{E}_i$ and $\mathcal{E}_j$ are integral expressions and that $\mathcal{E}_x$ and $\mathcal{E}_y$ are floating-point expressions. Further, take o to mean any of the arithmetic operations: + − · / or *. Then,

$\mathcal{E}_i$ o $\mathcal{E}_j$ is an integral expression, and

$\mathcal{E}_i$ o $\mathcal{E}_y$
$\mathcal{E}_x$ o $\mathcal{E}_j$ } are floating-point expressions.
$\mathcal{E}_x$ o $\mathcal{E}_y$

In general, if either expression is floating-point, then the result of combining them will be floating-point; if both expressions are integral, then the combination is also integral.

When mixed values are combined by the operations + − · and / , the compiler provides the program to convert the integral value to its corresponding floating-point form. The actual computation is done in floating-point. Exponentiation is usually performed by a routine taken from the library of the compiler. Separate entries to this routine are provided for each of the four combinations of integral and floating-point values.

If in a mixed combination the integer is a constant, the necessary conversion is performed at the time of compilation, resulting in no loss of efficiency in the object program. For example, if X is a floating-point variable, the expression X + 1 will be compiled as though the user had written X + 1.0.

Arithmetic Combinations of Integers

As mentioned in CHAPTER II, integers may be no more than ten digits in length. In all arithmetic operations, digits are dropped from the most significant end of the answer to produce a ten-digit result. Thus

(734981)·(1000000) yields 4981000000;
5000000001 + 5000000001 yields 2.

Division of integers is unrounded. Thus,

3/2 yields 1; 7/11 yields 0; 41/3 yields 13.

Division by zero is undefined.

BOOLEAN EXPRESSIONS

Boolean quantities may be combined by means of 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 transform on the comparison to produce a truth value. Such a truth value may be used either to effect a change of program control, or (if the relation is enclosed in parentheses) to produce a Boolean value ***true*** or ***false***.

GENERAL FORMS:

First form:

$\mathcal{E}_1$ ρ $\mathcal{E}_2$

where $\mathcal{E}_1$ and $\mathcal{E}_2$ are arithmetic expressions, and ρ is a relational operator. This relation has the value ***true*** if $\mathcal{E}_1$ does indeed stand in the relation ρ to $\mathcal{E}_2$; it is otherwise ***false***. It is used only in control statements; (see IF, UNTIL, and EITHER IF).

Second form:

($\mathcal{E}_1$ ρ $\mathcal{E}_2$)

This produces the Boolean value one (***true***) if $\mathcal{E}_1$ ρ $\mathcal{E}_2$ is ***true***, and produces the value zero (***false***) otherwise. Since the result is Boolean, it may be combined with any of the other Boolean operators previously discussed. The relational operators employed in this compiler are GTR, GEQ, EQL, LEQ, LSS, and NEQ. Their significance is indicated in the following table.

EXPRESSION†	CONVENTIONAL MATHEMATICAL NOTATION	MEANING
$\mathcal{E}_1$ GTR $\mathcal{E}_2$	$\mathcal{E}_1 > \mathcal{E}_2$	greater than
$\mathcal{E}_1$ GEQ $\mathcal{E}_2$	$\mathcal{E}_1 \geq \mathcal{E}_2$	greater than or equal to
$\mathcal{E}_1$ EQL $\mathcal{E}_2$	$\mathcal{E}_1 = \mathcal{E}_2$	equal to
$\mathcal{E}_1$ LEQ $\mathcal{E}_2$	$\mathcal{E}_1 \leq \mathcal{E}_2$	less than or equal to
$\mathcal{E}_1$ LSS $\mathcal{E}_2$	$\mathcal{E}_1 < \mathcal{E}_2$	less than
$\mathcal{E}_1$ NEQ $\mathcal{E}_2$	$\mathcal{E}_1 \neq \mathcal{E}_2$	not equal to

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

Within the context of this compiler, two numbers are equal if the quantities which are their internal machine representations are identical. Note, first, with regard to floating-point quantities, if A−B EQL 0, it is not necessarily the case that A EQL B; and second, that zeros are equal to each other regardless of sign.

EXAMPLES:

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

The compiler permits arithmetic operations to be performed on quantities which are not of the same type. In similar fashion, relational operators may be used to compare quantities which are not of the same type. If a floating-point quantity is to be compared to an integral quantity, the integer will be converted to its corresponding floating-point form prior to the comparison. If the integer is a constant, the conversion occurs during compilation.

Boolean Operations

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

The expression NOT P is ***true*** whenever P itself is ***false***; it is ***false*** whenever P is ***true***.

The expression P AND Q is ***true*** if and only if both P and Q are ***true***. If either P or Q is ***false***, then P AND Q is also ***false***.

The expression P OR Q is ***true*** if either P or Q or both are ***true***. P OR Q is ***false*** only when both P and Q are ***false***.

The expression P IMPL Q is ***true*** whenever either Q is ***true*** or both P and Q are ***false***. If P is ***true*** and Q is ***false***, then P IMPL Q is ***false***.

The expression P EQIV Q is ***true*** if P and Q are both ***true*** or both ***false***. If either P is ***true*** and Q is ***false*** or P is ***false*** and Q is ***true***, then P EQIV Q is ***false***.

These definitions are summarized in the following table, in accordance with this representation of ALGOL.

P	Q	NOT P	P AND Q	P OR Q	P IMPL Q	P EQIV Q
false	false	true	false	false	true	true
true	true	false	true	true	true	true
true	false	false	false	true	false	false
false	true	true	false	true	true	false

Construction of Boolean Expressions

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

In addition, if $\mathcal{E}_1 \rho \mathcal{E}_2$ is an arithmetic relation, and $\mathcal{B}_1$ and $\mathcal{B}_2$ are any Boolean expressions, then each of the following is also a Boolean expression:

$(\mathcal{E}_1 \rho \mathcal{E}_2)$	$\mathcal{B}_1$ OR $\mathcal{B}_2$	$(\mathcal{B}_1)$
NOT $\mathcal{B}_1$	$\mathcal{B}_1$ EQIV $\mathcal{B}_2$	
$\mathcal{B}_1$ AND $\mathcal{B}_2$	$\mathcal{B}_1$ IMPL $\mathcal{B}_2$	

Precedence of Boolean Operations

Conventions for the order of precedence of Boolean operations are not so 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; OR *will be executed before* IMPL; *and* IMPL *will be executed before* EQIV.

The expression P IMPL Q IMPL R is ambiguous; parentheses should be used to express the exact meaning desired.

EXAMPLES:

```
NOT(P AND Q) OR R IMPL P OR NOT Q
NOT (NOT P) EQIV 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) OR (N GTR 40)
R AND S OR (F(Z) EQL 4)
(M.N(R - 2) + 4 LSS TAN (BETA - ALPHA)) OR FLAG
(U.SINH(M) GTR M7) EQIV (V.COSH(M) GTR M12)
```

Any Boolean expression may appear in an arithmetic expression, where it will in all respects behave as if it were an integer taking on the values zero or one. In such a case, Boolean operations will be executed prior to arithmetic operations, unless parentheses have been used to specify otherwise.

EXAMPLES:

```
G - 0.3N.(D LSS 300)
A + V.NOT B1 OR B2
```

ASSIGNMENT STATEMENTS

GRAMMAR OF STATEMENTS

COMPOUND STATEMENTS

STATEMENT LABELS

IV. . .

statements

THE *statement*, $\mathcal{S}$, 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:

$\mathcal{V} = \mathcal{E}$

where $\mathcal{V}$ is a variable and $\mathcal{E}$ is an expression. Note that the symbol = is used in a special sense in this compiler to signify the process of substitution. 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 evaluated by a compiled program, which then assigns the value of $\sqrt{K}$ to the variable X.

Arithmetic Assignment Statements

If the variable $\mathcal{V}$ in $\mathcal{V} = \mathcal{E}$ is of integer or floating-point type, then we have an arithmetic assignment statement. If $\mathcal{V}$ is an integer and $\mathcal{E}$ is floating-point, then the value of $\mathcal{E}$ will be converted to integer form (truncating any fractional part) before the assignment is made. If $\mathcal{V}$ is floating-point and $\mathcal{E}$ is integral, then the value of $\mathcal{E}$ will be converted to the corresponding floating-point number. If $\mathcal{E}$ is Boolean, it is treated as if it were integral.

EXAMPLES:

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

Boolean Assignment Statements

If the $\mathcal{V}$ in $\mathcal{V} = \mathcal{E}$ is a Boolean variable, we then have a Boolean assignment statement. In this case, *the expression $\mathcal{E}$ must be Boolean.*

EXAMPLES:

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

Generalized Assignment Statement

GENERAL FORM:

$\mathcal{V}_1 = \mathcal{V}_2 = \cdots = \mathcal{V}_n = \mathcal{E}$

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 of mixed type, then conversion of type will be performed; e.g., assume X, Y, and $\mathcal{E}$ are floating, and I is integer. Then the statement

X = I = Y = $\mathcal{E}$

will cause $\mathcal{E}$ to be truncated to an integer before storing into I, and this truncated result floated before storing into X. Thus, in this example, X = I = Y = $\mathcal{E}$, X = Y = I = $\mathcal{E}$, and I = X = Y = $\mathcal{E}$ may all give different results when $\mathcal{E}$ is floating.

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 *statements must be separated by semicolons.*

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

GENERAL FORM:

$\cdots \mathcal{S}; \mathcal{S} \cdots$

where the symbol $\mathcal{S}$ represents any statement. 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 $\mathcal{S}_1; \mathcal{S}_2; \cdots; \mathcal{S}_n$ END

where $\mathcal{S}_1$ through $\mathcal{S}_n$ are statements. The words BEGIN and END serve as opening and closing 'statement parentheses.' Indeed, the symbols '(' and ')' may be substituted for the words BEGIN and END with no change in meaning.

Throughout this description of the compiler, unless the contrary is specifically stated, the word 'statement' and the symbol $\mathcal{S}$ 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 V.

EXAMPLES:

```
BEGIN U = -B/2A; V = SQRT(U*2 - C/A);
R1 = U + V; R2 = U - V END

BEGIN S = SIN(THETA); C = COS(THETA);
XI = C.X + S.Y; ETA = -S.X + C.Y END

(S = A(I,J); A(I,J) = A(J,I); A(J,I) = S)
```

Statement Labels

It is often necessary to attach a name to a statement. This name is called a *statement label* $\mathcal{L}$. A statement label may consist of an identifier or of an integer. (Leading zeros of an integer used as a statement label are without meaning to the compiler—the statement labels 13 and 013 are in all ways equivalent.)

GENERAL FORMS:

First form:

$\mathcal{I}$.. $\mathcal{S}$

Second form:

$\mathcal{N}_I$.. $\mathcal{S}$

where $\mathcal{I}$ is an identifier, $\mathcal{N}_I$ is an integer, and $\mathcal{S}$ is any statement.

EXAMPLES:

```
START..SUM = 0
LEGENDRE..P(N) = ((2N - 1) P(N - 1) - (N-1)P(N - 2))/N
ROTATE..BEGIN S = SIN(THETA); C = COS(THETA);
XI = C.X + S.Y; ETA = -S.Y + C.Y END
27..BETA = ARCTAN(HORIZ/VERTL) - ALPHA
```

When labeling a compound statement, the programer may repeat the statement label after the word END. This may be done for readability of the print-out produced during compilation; the compiler itself makes no use of the information.

GENERAL FORM:

£..BEGIN S_1; ⋯; S_n END £

EXAMPLES:

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

In those cases where it is necessary during the running of a program to transfer from a point in a BEGIN ⋯ END clause to a point just before the word END, a labeled dummy statement is employed. This statement label, which does not in itself produce any action, takes the form of an identifier followed by two periods, directly preceding the word END which terminates the group of statements. An example of this is shown on page 6-5, SEARCH OF A RECTANGULAR GAME FOR A SADDLE POINT.

GENERAL FORM:

BEGIN S_1; S_2; ⋯; S_n; £..END

It is sometimes necessary to introduce a section of machine-language coding into the compiled program, which will then act in all respects like a statement. To do this, one employs the declarator EXTERNAL STATEMENT in the program.

GENERAL FORM:

EXTERNAL STATEMENT £

The identifier £ serves as the label of the EXTERNAL STATEMENT.

The definition of the statement, i.e., the machine-language program itself, appears after the FINISH declarator of the symbolic program. (See APPENDIX E, *Construction of Machine-Language Programs*).

TYPE
ARRAY
COMMENT
FINISH

V . . .

basic declarations

THE DECLARATIONS OF TYPE—FLOATING, REAL, INTEGER, and BOOLEAN are defined in this chapter, together with the ARRAY, COMMENT, and FINISH declarations. These do not exhaust the entire set of declarations available to the programer; however, the others constitute separate subjects in themselves and are therefore reserved for later chapters.

Declarations determine how the compiled program will treat certain of its elements. It is thus necessary to precede the use of an element with such a declaration.

DECLARATIONS OF TYPE

Declarations of type are used to indicate that a specified set of identifiers represent quantities of a given type (floating-point, integer, or Boolean). By the use of prefixes, entire classes of identifiers are declared to be of a given type. In addition, it is possible to declare that a variable not appearing in any declaration of type is of a given type.

Construction of Declarations of Type

GENERAL FORM:

FLOATING 𝔗𝔏
REAL 𝔗𝔏
INTEGER 𝔗𝔏
BOOLEAN 𝔗𝔏

where 𝔗𝔏 is a *type list* to be defined below. These statements declare the identifiers given in 𝔗𝔏 to be of floating-point, integer, and Boolean types. (FLOATING and REAL produce equivalent results in the compiler.) A *type list* consists of a sequence of entries separated by commas. Possible entries include identifiers, identifiers followed by blank subscripts, and prefixes.

EXAMPLE:

```
INTEGER I, J, K, L, Z, GCD(,), TABLE( )
```

(Note that the use of a pair of parentheses following an identifier in a declaration of type has no effect on the compiled program; they are there only for the convenience of the programer.)

EXAMPLE:

```
BOOLEAN (SW1, SW2, FLAG, TOGGLE(,,))
```

The Use of Prefixes

If desired, one may put a prefix into a type list rather than use an identifier. A *prefix* consists of an identifier followed by three periods. The maximum number of characters is five, for an identifier used for this purpose.

GENERAL FORM:

𝔤...

EXAMPLE:

```
MQR4...
```

The appearance of this prefix in a declaration of type means that any variable, function, or array, the identifier of which has MQR4 as its first four characters, and which is not otherwise declared, is of the specified type.

It is possible for prefixes to introduce apparent ambiguities. Consider, for example,

```
FLOATING ABCD, AB4; INTEGER AB...;
BOOLEAN ABC...
```

What are the types of AB13, ABCD, ABCDEF, AB4, and AB5? The rule governing this situation is: *Unless specifically indicated in a type list, an identifier is matched against the longest applicable prefix.* Thus, the above identifiers are of integer, floating, Boolean, floating, and integer types, respectively.

Declaration by Default

The word OTHERWISE may be written in lieu of a type list. This form indicates that any name of a variable, array, or function not specifically declared and not matching any of the prefixes is to be of the type denoted by this declaration. If no such statement is given, any undeclared variable, array, or function will be assumed to be floating-point. This construction may be called *declaration by default.*

EXAMPLES:

```
BOOLEAN SW, P,Q,R; FLOATING X,Y,Z,F( );
  INTEGER OTHERWISE

INTEGER I, J, K, N, M..., G; BOOLEAN OTHERWISE
```

To repeat the remark made in the introduction of this chapter: *The type of an identifier must be declared before that identifier is used in any other statement. If an identifier is used prior to a declaration of type it is declared, by default, as* FLOATING.

THE ARRAY DECLARATION

The ARRAY declaration provides a means of referring to a collection of numbers 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.

Arrays in this compiler are restricted to those of rectangular construction in n-dimensional space.

If the identifier of an array is declared in a declaration of type, then that declaration of type must precede the ARRAY declaration.

Construction of ARRAY Declarations

An array must have been described by an ARRAY declaration prior to the use of any variable with subscripts which represents an element of that array.

GENERAL FORM:

ARRAY $\mathfrak{L}\mathfrak{g}$, $\mathfrak{L}\mathfrak{g}$, $\sim$, $\mathfrak{L}\mathfrak{g}$

where $\mathfrak{L}\mathfrak{g}$'s are list items of the array declarator list. These list items take on two general forms:

First form:

$\mathfrak{g}\ (\mathfrak{N}_1, \sim, \mathfrak{N}_q)$

Second form:

$\mathfrak{g}\ (\mathfrak{N}_1, \sim, \mathfrak{N}_q) = (\mathfrak{M}, \mathfrak{M}, \mathfrak{M}, \sim, \mathfrak{M})$

Both of these forms declare $\mathfrak{g}$ to be an array of q dimensions. Each dimension contains the number of elements given by the corresponding value of $\mathfrak{N}$; hence the value of $\mathfrak{N}$ is the maximum which a corresponding subscript expression may assume.

The second form is used to set initial values for the elements of the array at load time; see *Filling an Array*, below.

EXAMPLE:

```
ARRAY (M(3,4), CHAR (6,6,6), VECTOR (100))
```

This declaration reserves twelve cells in storage for the two-dimensional array M, 216 cells for the three-dimensional array CHAR, and 100 cells for the one-dimensional array VECTOR.

At the programer's option, the list of arrays being declared may be enclosed in parentheses to improve the readability of the symbolic program. Such use of these parentheses will have no effect on the compilation.

Filling an Array

An item in an ARRAY declaration may have appended to it a list of values to be assigned at the beginning of computation to the elements of the array. Referring to the second form, above:

Second form:

$\mathfrak{g}\ (\mathfrak{N}_1, \sim, \mathfrak{N}_q) = (\mathfrak{M}, \mathfrak{M}, \mathfrak{M}, \sim, \mathfrak{M})$

The quantities $\mathfrak{M}$ are constants (with their respective signs) which are placed in the positions of the array $\mathfrak{g}$. (The compiler will, if necessary, convert these constants at the time the program is compiled to agree in type with that of the identifier $\mathfrak{g}$.)

EXAMPLE:

```
ARRAY Q(3,2) = (7.3, 9.1, 4, 127.3, +4.19, −2.2)
```

Assuming Q has been previously declared to be floating, this declaration will cause the matrix Q(I, J)

$$\begin{bmatrix} 7.3 & 9.1 \\ 4.0 & 127.3 \\ 4.19 & -2.2 \end{bmatrix}$$

to be available in memory when the compiled program is loaded.

It is not necessary to fill up the entire array in this manner. Cells to which no constant is assigned are cleared to zero at the time the program is loaded from tape.

Referring again to the second form above, the constants $\mathfrak{M}$ are placed in the array $\mathfrak{g}$ in the following order:

The first subscript is (1, 1, ..., 1, 1). For each succeeding value of $\mathfrak{M}$ the rightmost subscript is advanced by one. After the rightmost subscript reaches its maximum value, $\mathfrak{N}_q$, it is reset to one, and at the same time the subscript to its left is advanced by one.

In similar fashion, the other subscripts are advanced, the subscript in the $(i-1)$th position being increased by one at the same time that the ith subscript is reset.

These cycles continue until all data have been stored, or until all subscripts have reached their respective maximum values.

Assuming an array $A(n_1, n_2, ..., n_{q-1}, n_q)$, this results in the following sequence of subscripts:

CYCLE	SUBSCRIPT
First	1, 1, ..., 1, 1 1, 1, ..., 1, 2 . . . . . . 1, 1, ..., 1, n_q
Second	1, 1, ..., 2, 1 1, 1, ..., 2, 2 . . . . . . 1, 1, ..., 2, n_q
Third	1, 1, ..., 3, 1 . . .
$(n_1 \cdot n_2 \cdot \ ... \ \cdot n_{q-1} - 1)$th	$n_1, n_2, ..., n_{q-1} - 1, 1$ $n_1, n_2, ..., n_{q-1} - 1, 2$ . . . . . . $n_1, n_2, ..., n_{q-1} - 1, n_q$
$(n_1 \cdot n_2 \cdot \ ... \ \cdot n_{q-1})$th	$n_1, n_2, ..., n_{q-1}, 1$ $n_1, n_2, ..., n_{q-1}, 2$ . . . . . . $n_1, n_2, ..., n_{q-1}, n_q$

THE COMMENT DECLARATION

The COMMENT declaration allows the programer to include any clarifying remarks, identifying symbols, etc., in the printed compilation. The COMMENT declaration 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 semicolon, a semicolon obviously cannot be used within the string of characters.

GENERAL FORM:

COMMENT $s_;$

where $s_;$ is any string of characters *not containing a semicolon.*

EXAMPLE:

```
COMMENT SMOOTH FIELD DATA AND REDUCE TO
  STANDARD FORM
```

There is one restriction on the use of the COMMENT declaration. *It must not be the last statement of a compound statement*; that is, a COMMENT statement must not be terminated by an 'END' or a ')'. Only a ';' may follow the comment.

THE FINISH DECLARATION

The FINISH declaration defines the end of the program being compiled, and terminates the compilation. A FINISH declaration must appear as the last statement in any program and may appear nowhere else in the program. The semicolon following a FINISH declaration is essential; it may not be omitted.

GENERAL FORM:

FINISH;

EXAMPLE:

```
FINISH;
```

(See page A-2 for the manner in which the compiler treats the FINISH declaration.)

TRANSFER OF CONTROL
TERMINATION OF COMPUTATION
CONDITIONAL EXECUTION

VI . . .

basic control statements

THIS CHAPTER deals with the means of expressing the 'flow of control' of an algorithm which has been described in compiler language. The order of evaluation of equations is as important to the description of an algorithm as are the equations themselves. Experience has shown that there is a relatively small number of constructions which commonly appear in the description of algorithms. This group of constructions has been included in the compiler language.

The basic control statements provide the abilities:

First, to transfer control to another part of the problem (the GO TO and SWITCH statements);

second, to terminate computation (the STOP statement);

third, to execute statements contingent on given criteria (the IF and alternative statements); and

fourth, to control iterative processes (the FOR and UNTIL statements).

TRANSFER OF CONTROL

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 £

where £ is a statement label.

The statement with the label £ will be executed immediately after the GO TO statement. The word TO is redundant and may be omitted.

EXAMPLES:

```
GO TO START
GO A16
GO TO 14
GO POGO
```

The SWITCH Statement

An extension of the GO TO statement is provided by the SWITCH statement. The SWITCH statement uses the value of an expression to determine transfer of control to one of a list of several statement labels.

GENERAL FORM:

SWITCH $\mathcal{E}$, ($£_1$, $£_2$, $£_3$, ..., $£_n$)

where $\mathcal{E}$ is an arithmetical expression and $£_1$ through $£_n$ are statement labels.

The action of a SWITCH statement is as follows:

Let i equal the integral part of the expression $\mathcal{E}$; then,

If $i = 0$, the SWITCH statement has no effect, and control continues in sequence.

If $| i | \leq n$, then a transfer of control is made to the statement the label of which is in the ith position in the list.

If $| i | \geq n + 1$, then the action of the SWITCH statement is undefined.

EXAMPLES:

```
SWITCH Y + 2, (A1, A2, A3)
```

If Y + 2 = 0, no transfer occurs.
If Y + 2 = 1, transfer to the statement labeled A1.
If Y + 2 = 2, transfer to the statement labeled A2.
If Y + 2 = 3, transfer to the statement labeled A3.

```
SWITCH 3I + J, (XA, XB, XC, YA, YB, YC)
SWITCH MOD(K,4) + 1, (ALPHA, BETA, GAMMA, DELTA)
```

TERMINATION OF COMPUTATION

The STOP Statement

The STOP statement serves to indicate the end of operation or a temporary halt in a compiled program. (Computation is resumed with the next statement in sequence

when the START lever is depressed.) If desired, a STOP statement may be accompanied by an expression the value of which is displayed in the A register when the computer stops.

GENERAL FORMS:

First form:
STOP

Second form:
STOP $\mathcal{E}$

where $\mathcal{E}$ is any expression.

In the case of the first form, the computer stops with the contents of the A register undefined; in the case of the second form, the value of the expression $\mathcal{E}$ is found in the A register.

EXAMPLES:

```
STOP
STOP 4241535362
STOP ANSWER(J)
```

CLAUSES

The GO TO, SWITCH, and STOP statements discussed thus far are by themselves complete statements and depend in no way on other statements to complete their meaning. The remainder of this chapter will discuss statements which bear an analogy to the dependent clauses of a natural language. In all cases, these clauses affect the behavior of the statement (or compound statement) which follows them.

A construction consisting of one or more clauses $\mathcal{C}$ followed by a statement $\mathcal{S}$,

$\mathcal{C}_1; \mathcal{C}_2; \mathcal{C}_3; \sim; \mathcal{S}_n$

is in itself a compound statement, requiring no additional punctuation.

Of course, if a clause is to affect the behavior of several statements, those statements must be grouped together as a compound statement:

$\mathcal{C}$; BEGIN $\mathcal{S}_1; \mathcal{S}_2; \mathcal{S}_3; \sim; \mathcal{S}_n$ END

Examples of these constructions will be given in context below.

CONDITIONAL EXECUTION

The IF Statement

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

GENERAL FORMS:

First form:
IF $\mathcal{B}$; $\mathcal{S}$

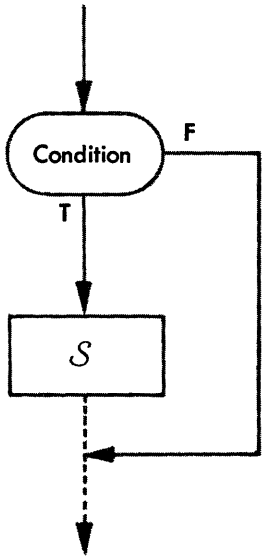
where $\mathcal{B}$ is a Boolean expression—a 'condition'—and $\mathcal{S}$ is any statement.

Second form:

IF $\mathcal{E}_1 \rho \mathcal{E}_2$; $\mathcal{S}$

where $\mathcal{E}_1$ and $\mathcal{E}_2$ are arithmetical expressions, ρ is a relational operation, and $\mathcal{S}$ is any statement.

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



First form:

If the Boolean expression $\mathcal{B}$ is ***true***, the statement $\mathcal{S}$ is executed; if $\mathcal{B}$ is ***false***, $\mathcal{S}$ is skipped over. In either case, control continues in sequence following $\mathcal{S}$.

EXAMPLES:

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

Second form:

The most common form of condition which appears in an algorithm is a simple relation between the magnitudes of two arithmetical quantities. While this situation is certainly provided for under the first form (see the first two examples above), the slightly more concise second form is also allowed. In those cases where the second form is applicable and is used, the result will be the compilation of a significantly more efficient object program. The use of the second form is recommended wherever possible.

EXAMPLES:

```
IF X*2 GTR 7; STOP
IF I EQL J; A(I,J) = 1
IF I EQL IX; SWITCH IX, (A,B,C)
IF ABS(TERM) LSS EPSILON; GO OUT
IF TGL4; IF Z GTR X + Y*2 − 4; BEGIN Z = Y − 1;
  X = Y/(U + Y); GO LOOP4 END
```

The Alternative Statement

An extended form of the IF statement is provided by the *alternative statement.* A sequence of conditions is examined—in order—until one is found which is satisfied. A statement associated with that particular condition is then executed; the remainder of the alternatives are ignored. An option is provided for indicating a statement to be executed in the event that none of the sequence of conditions is satisfied.

GENERAL FORMS:

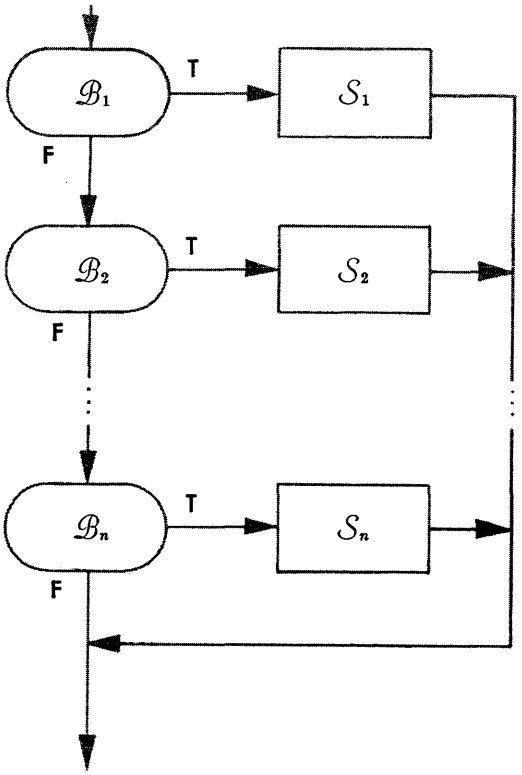
First form:

EITHER IF $\mathcal{B}_1$; $\mathcal{S}_1$; OR IF $\mathcal{B}_2$; $\mathcal{S}_2$; ⋯;
OR IF $\mathcal{B}_n$; $\mathcal{S}_n$ END

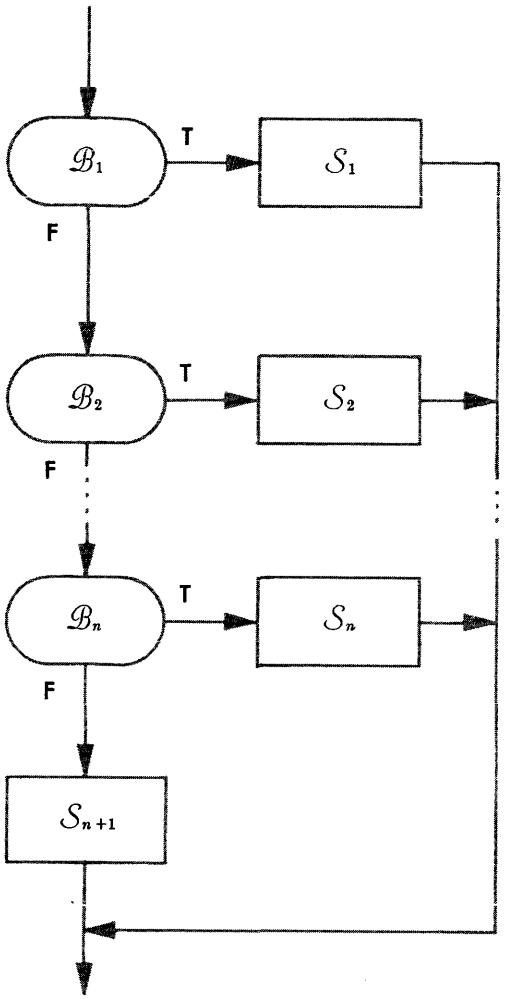
Second form:

EITHER IF $\mathcal{B}_1$; $\mathcal{S}_1$; OR IF $\mathcal{B}_2$; $\mathcal{S}_2$; ⋯;
OR IF $\mathcal{B}_n$; $\mathcal{S}_n$; OTHERWISE; $\mathcal{S}_{n+1}$

The following flow charts will serve to explain these two statements more precisely. For the first form we have:



For the second form we have:

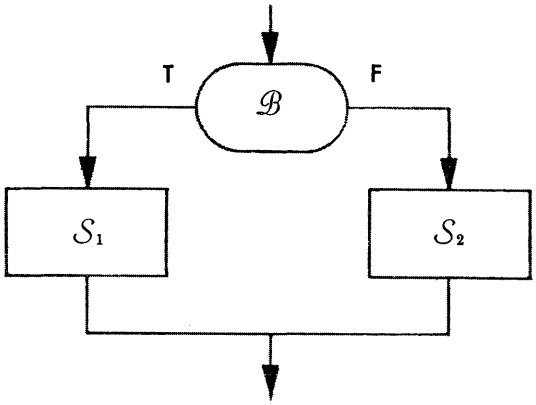


Any of the conditions marked $\mathcal{B}$ may be replaced by the simpler form, $\mathcal{E}_1 \rho \mathcal{E}_2$, whenever desired.

In the case of the second form, no OR IF clauses need be used. If no OR IF clauses are used, the alternative statement becomes

EITHER IF $\mathcal{B}$; $\mathcal{S}_1$; OTHERWISE; $\mathcal{S}_2$

which expresses the very common construction:



EXAMPLE:

```
COMMENT EVALUATE POLYNOMIAL
USING RECURSION RELATION EMPLOYING PREVIOUS
VALUES WHEN POSSIBLE. N IS ORDER OF POLYNOMIAL
AND X IS ARGUMENT;
EITHER IF N EQL 0; BEGIN M = 1.0**40;
PN1 = 1.0 END;
OR IF N EQL 1; BEGIN M = 3; Z = X + X;
R = X + Z; PN2 = 1; PN1 = X END;
OR IF (X + X EQL Z) AND (N GEQ M − 1);
GO TO RECURSE;
OTHERWISE; BEGIN PN2 = 1; PN1 = X;
Z = X + X; R = X + Z; M = 3;
RECURSE.. BEGIN N = N + 1; FOR M = (M, 1, N);
BEGIN R = R + Z; PN = (R.PN1 − (M − 1)PN2)/M;
PN2 = PN1; PN1 = PN END END RECURSE END;
POLYNOMIAL = PN1
```

CONTROL OF ITERATIONS

The UNTIL Statement

The UNTIL statement is used primarily to provide control of iterative processes where escape from the loop depends upon a result calculated within the loop.

GENERAL FORMS:

First form:

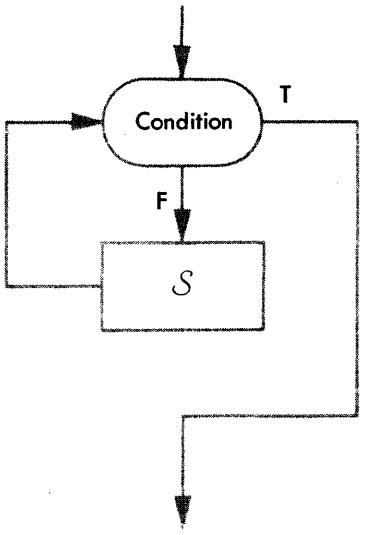
UNTIL $\mathfrak{B}$; $\mathfrak{S}$

Second form:

UNTIL $\mathfrak{E}_1$ ρ $\mathfrak{E}_2$; $\mathfrak{S}$

where $\mathfrak{B}$ is any Boolean expression, $\mathfrak{E}_1$ ρ $\mathfrak{E}_2$ is an arithmetic relation, and $\mathfrak{S}$ is any statement or statement group.

The action of the UNTIL statement is described graphically by means of the following flow chart:



First form:

The statement $\mathfrak{S}$ is executed repetitively until the Boolean condition is satisfied; control then continues in sequence. If $\mathfrak{B}$ is satisfied initially, $\mathfrak{S}$ will not be executed at all.

EXAMPLES:

```
COMMENT TABLE LOOK-UP;
I = 1; UNTIL T(I) GEQ ARGMT; I = I + 1
```

```
UNTIL (L − LPRIME LSS TOLERANCE) AND
  (V LSS 0.01); BEGIN V = V*2; LPRIME = L;
  L = ITER (L,V) END
```

```
COMMENT SUM SERIES FOR E*−X;
N = 2; E = 1; T = 1; UNTIL (ABS(T) LSS 1**−6) OR
  (N GTR 30); BEGIN T = −T.X/N; E = E + T;
  N = N + 1 END
```

Second form:

As in the case of the IF statement, the UNTIL statement is provided with an alternate form to produce a more efficient object program in those cases where the condition to be tested consists of a simple relation between two arithmetic quantities.

EXAMPLES:

```
COMMENT SEARCH FOR ROOT OF F( );
FA = F(A); UNTIL ABS (B − A) LSS EPS;
BEGIN U = (A + B)/2; EITHER IF FA.F(U) GTR 0; A = U;
OTHERWISE; B = U END
```

```
COMMENT ITERATED TRAPEZOIDAL INTEGRATION;
H = (B − A); I = (F(A) + F(B))/2; J = 0;
UNTIL I − 2J LSS 0.003H;
  BEGIN Q = H; J = I; H = H/2; X = A + H;
  UNTIL X GTR B;
  BEGIN I = I + F(X); X = X + Q END END
```

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 $\mathfrak{V}$ = $\mathfrak{IL}$; $\mathfrak{S}$

where $\mathfrak{V}$ is a variable, $\mathfrak{IL}$ is an iteration list, and $\mathfrak{S}$ is any statement or statement group.

The iteration list describes the sequence of values that the variable $\mathfrak{V}$ is to assume. The statement $\mathfrak{S}$ will be executed for each of these values. After the iteration list has been exhausted, the statement following $\mathfrak{S}$ will be executed.

The most common form that an iteration assumes is a triplet of expressions separated by commas and enclosed in parentheses.

First form:

$(\mathcal{E}_I, \mathcal{E}_S, \mathcal{E}_T)$

where $\mathcal{E}_I$, $\mathcal{E}_S$, and $\mathcal{E}_T$ are arithmetical expressions.

In this case, the FOR statement takes on the form:

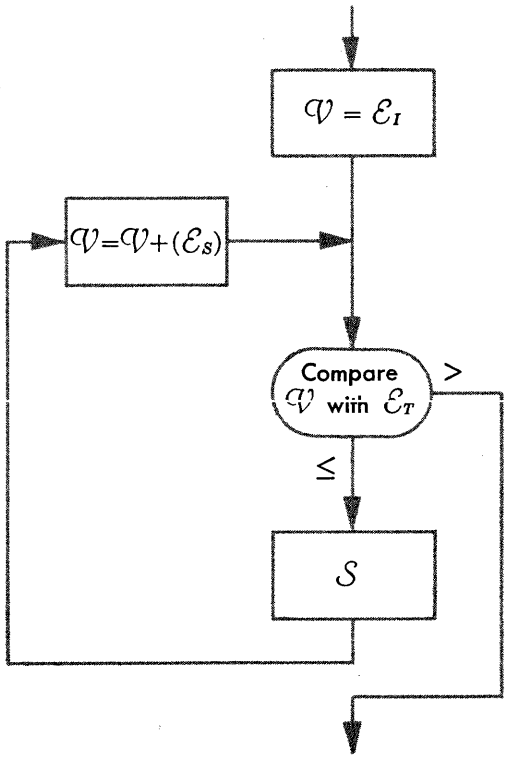
FOR $\mathcal{V} = (\mathcal{E}_I, \mathcal{E}_S, \mathcal{E}_T)$; $\mathcal{S}$

If the first character of $\mathcal{E}_S$ is not a minus sign, then the form is equivalent to the simpler statements:

$\mathcal{V} = \mathcal{A} = \mathcal{E}_I$; $\mathcal{L}$.. IF $\mathcal{A}$ LEQ $\mathcal{E}_T$;

BEGIN $\mathcal{S}$; $\mathcal{V} = \mathcal{A} = \mathcal{V} + \mathcal{E}_S$; GO TO $\mathcal{L}$ END

where $\mathcal{A}$ represents the value of the induction variable $\mathcal{V}$.



In the case that the first character of $\mathcal{E}_S$ is a minus sign, the FOR statement is equivalent to:

$\mathcal{V} = \mathcal{A} = \mathcal{E}_I$; $\mathcal{L}$.. IF $\mathcal{A}$ GEQ $\mathcal{E}_T$;

BEGIN $\mathcal{S}$; $\mathcal{V} = \mathcal{A} = \mathcal{V} + \mathcal{E}_S$; GO TO $\mathcal{L}$ END

and the preceding flow chart holds if we replace $>$ by $<$, and $\leq$ by $\geq$.

Note that if the test fails initially (i.e., $\mathcal{E}_I > \mathcal{E}_T$ in the first case, or $\mathcal{E}_I < \mathcal{E}_T$ in the second), the triplet is considered vacuous, and the statement $\mathcal{S}$ will not be executed at all.

On exit from the FOR statement, the value of the induction variable $\mathcal{V}$ is that which it has when the test first failed. In consequence, the use of a GO or a GO TO statement to transfer to any labeled statement included in the statement $\mathcal{S}$ within such a FOR statement may produce anomalous results.

EXAMPLES:

```
COMMENT EVALUATE INNER PRODUCT OF U( ) AND V( );
  DOT = 0; FOR I = (1,1,N);
  DOT = DOT + U(I).V(I)

COMMENT SEARCH RECTANGULAR GAME FOR
  SADDLE POINT;
FOR I = (1,1,M); BEGIN L = 0;
  FOR J = (1,1,N); IF A(I,J) GTR L;
    BEGIN L = A(I,J); T = J END;
  FOR K = (1,1,M); IF A(K,T) LSS L; GO AGAIN;
  GO FOUND; AGAIN.. END; GO NONE

COMMENT SOLVE EQUATIONS A(N X (N + 1)) FOR X( );
FOR K = (N + 1, -1, 1); BEGIN
  FOR I = (1,1,N); X(I) = A(I,1);
  FOR J = (2,1,K); BEGIN
    D = A(1,J)/X(1);
    FOR I = (2,1,N); A(I - 1, J - 1) = A(I,J) - X(I).D;
    A(N,J) = D END END
```

The second form that an iteration list may assume is a list of expressions separated by commas.

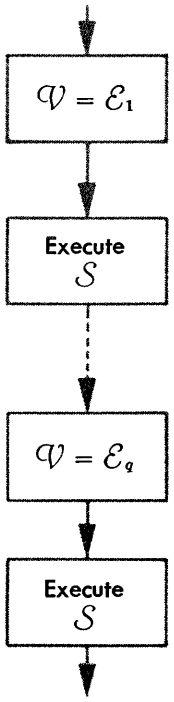
Second form:

$\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3, \ldots, \mathcal{E}_q$

In this case, the FOR statement appears as

FOR $\mathcal{V} = \mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3, \ldots, \mathcal{E}_q$; $\mathcal{S}$

The behavior of this statement may be clarified by the following flow chart:



That is, the variable $\mathfrak{v}$ is successively given the values of $\mathfrak{e}_1$, $\mathfrak{e}_2$, and so on through $\mathfrak{e}_q$. The statement $\mathfrak{s}$ is executed once for each value which $\mathfrak{v}$ assumes.

EXAMPLES:

```
FOR PRIME = 2,3,5,7,11,13,17; s
FOR X = 0, 0.1, 0.5, 1.0, 5.0, 10.0; s
```

Third form:

The *third form* of an iteration list is actually a combination of the first two: triplets of expressions which appear in the first form may be used as members of the list of the second form. The sequence of values which results is the expected one.

EXAMPLES:

```
FOR Z = (0,1,10), (15,5,50), (100,50,1000), 5000,10000; s
```

This statement would cause $\mathfrak{s}$ to be executed for Z = 0, 1, 2, ..., 9, 10, 15, 20, 25, ..., 50, 100, 150, 200, 250, ..., 900, 950, 1000, 5000, and 10000.

```
FOR I = (1,1,N); FOR J = (1,1,I - 1), (I + 1,1,N);
  A(I,J) = A(I,J)/A(I,I)
```

This statement will divide off-diagonal elements of each row of matrix A(,) by the diagonal element of that row. Note that the first triplet of the second FOR clause is vacuous when I = 1; the second is vacuous when I = N.

SUBROUTINES
FUNCTIONS
INTRINSIC FUNCTIONS
PROCEDURES
EXTERNAL PROCEDURES

VII . . .

subprograms

ONE OF THE MOST IMPORTANT aspects of the stored-program computing device is its ability to treat subprograms which may be executed from any point in the main program. The compiler language includes several methods of defining subprograms, each of which has its particular field of application.

The declarations SUBROUTINE, FUNCTION, and PROCEDURE will be discussed, as well as the ENTER statement, the RETURN statement, a variation of the assignment statement, and the procedure-call statement. These declarations and statements are peculiar to the definition and use of subprograms.

SUBROUTINES

The form of the subprogram which is conceptually the simplest consists merely of a compound statement, which may be executed on demand from any part of the remainder of the program without the necessity of rewriting the actual compound statement each time its particular effect is desired. For our purposes here, such a compound statement will be called a *subroutine.*

The SUBROUTINE Declaration

The SUBROUTINE declaration states that the following compound statement represents a subroutine.

GENERAL FORM:

SUBROUTINE $\mathcal{I}$; BEGIN S_1; S_2; ⋯; S_n END

where $\mathcal{I}$ is an identifier and S_1 through S_n are the statements which define the effect of the subroutine. The identifier becomes the subroutine label. In such a case, $\mathcal{I}$ is not a statement label in the usual sense although it may—at the programer's option—follow the word END, as may a label of any compound statement.

All the identifiers which appear in a subroutine have precisely the same meanings as those assigned to them outside the subroutine. This is what is meant when a subroutine is said to be *dependent* on the program in which it is defined.

The RETURN Statement

The compound statement which defines the subroutine is executed starting with its first component statement. One (or more) of the statements S_1 through S_n which compose the subroutine and which follow the SUBROUTINE declaration must be a RETURN statement. Computation within the subroutine proceeds until a RETURN statement is encountered.

GENERAL FORM:

RETURN

The RETURN statement causes control again to be resumed in sequence at that point at which the subroutine was called. *The* RETURN *statement is the only manner in which an exit from the subroutine may be effected.* 'Running off the end' of a subroutine will produce anomalous results.

A subroutine need not be defined prior to its use. Subroutines may be defined within other subroutines.

EXAMPLE:

```
SUBROUTINE EVALUATE; BEGIN U = 0; V = 0;
  FOR I = (1,1,N); FOR J = (1,1,N); BEGIN W = 0;
  FOR K = (1,1,N); W = W + X(I,K).Y(K,J); IF I EQL J;
  W = W - 1; U = U + ABS(W); V = MAX(V,ABS(W)) END;
  U = U/N*2; RETURN END EVALUATE
```

The ENTER Statement

The ENTER statement is used to initiate the execution of a subroutine (*to call* a subroutine).

GENERAL FORM:

ENTER $\mathcal{I}$

where $\mathcal{I}$ is the label of a subroutine.

EXAMPLE:

```
SUBROUTINE CHEBYSHEV;
BEGIN EITHER IF N EQL 0; (M = 1.0**40; PN1 = 1.0);
OR IF N EQL 1; (M = 2; Z = X + X; PN2 = 1; PN1 = X);
OR IF (X + X EQL Z) AND (N GEQ M); ENTER RECURSE;
OTHERWISE; BEGIN PN2 = 1; PN1 = X; Z = X + X; M = 2;
  ENTER RECURSE END;
CHEBY = PN1; RETURN;
SUBROUTINE RECURSE;
BEGIN FOR M = (M,1,N);
BEGIN PN = Z.PN1 - PN2; PN2 = PN1; PN1 = PN END;
  RETURN END END CHEBYSHEV
```

FUNCTIONS

Another sort of subprogram is that resulting from the FUNCTION declaration. The reader should keep in mind that the FUNCTION declaration is only one of several ways in which functions are made available to the program being compiled.

The FUNCTION Declaration

The FUNCTION declaration serves to define those functions of a particularly simple and common kind, which may be expressed by means of a single expression. *A function must be declared before it is used.*

GENERAL FORM:

FUNCTION $\mathcal{F}$ ($\mathcal{P}_1$, $\mathcal{P}_2$, ⁓, $\mathcal{P}_n$) = $\mathcal{E}$

where $\mathcal{F}$ is an identifier which is to be the name of the function, $\mathcal{P}_1$ through $\mathcal{P}_n$ are identifiers which serve as the parameters of the function, and $\mathcal{E}$ is the expression which defines the function.

Any well-formed expression $\mathcal{E}$ involving the identifiers $\mathcal{P}_i$ and any other identifiers appearing in the program may be used. Identifiers used as parameters of a FUNCTION declaration are independent of identifiers used elsewhere in the program, *even though the identifiers used as parameters are spelled in exactly the same way as the identifiers used in the program.* (This is subject to the provision that there is no conflict between the declarations of type for the identifiers used in the main program and the desired type for those identifiers, spelled in the same way, which are used as parameters.) The compiler treats all other identifiers appearing in $\mathcal{E}$ as if they were part of the main program.

The types (integer, floating, Boolean) of the parameters and the type of the value of the function itself are determined by the declarations of type in the same manner as are other identifiers.

EXAMPLES:

```
FUNCTION ROOT (A,B,C) = (-B + SQRT(B*2 - 4A.C))/2A
FUNCTION NORM (X,Y) = SQRT( (U.X*2 + V.Y*2)/(U + V))
FUNCTION NEGEXP(Z) = (1 + Z(0.2507213 + Z(0.0292732
  + 0.0038278Z)))*-4
FUNCTION ARCSINH(S) = LOG(S + SQRT(S*2 + 1))
FUNCTION STROKE(P,Q) = NOT (P AND Q)
```

Functions may be declared inside of subroutines.

Intrinsic Functions

There is a small group of functions called *intrinsic functions*, the definitions of which are a part of the compiler. Each of the intrinsic functions is discussed in turn and a tabular summary of them is given below.

MOD ($\mathcal{E}_1$, $\mathcal{E}_2$)

The function MOD requires two integer arguments. The value of the function is the integer obtained as the remainder when the first argument is divided by the second.

MAX ($\mathcal{E}_1$, ⁓, $\mathcal{E}_q$) and MIN ($\mathcal{E}_1$, ⁓, $\mathcal{E}_q$)

The functions MAX and MIN must have two or more arguments. The value of the function MAX will be the value of the largest of its arguments (algebraically); the value of the function MIN will be the value of the smallest of its arguments (algebraically).

The arguments may be either integer or floating-point expressions. If all of the arguments are integers, then the value of the function will be an integer. If any of the arguments is floating-point, then the result will also be floating-point.

SIGN ($\mathcal{E}$)

The function SIGN has a single argument. If this argument is positive, the result will be +1; if zero, the result is also zero; if negative, the result is −1. The result of the SIGN function will be of the same type as that of its argument.

ABS ($\mathcal{E}$)

The function ABS has a single argument. The result of the ABS function will be the absolute value of the argument and will be of the same type as its argument.

PCS ($\mathcal{E}$)

The function PCS is used for interrogating the PROGRAM CONTROL SWITCHES. Its value is Boolean in type and is ***true*** if the indicated PROGRAM CONTROL SWITCH is ON and ***false*** if OFF. The units digit of the argument of the PCS function indicates which of the PROGRAM CONTROL SWITCHES, 0 through 9, is to be interrogated. The argument may be either integral or floating-point. (If the MONITOR declaration is employed, the use of PCS(0) may be restricted. See page 10-3.)

INTRINSIC FUNCTIONS

Name and Description	Type of Function	Type of Argument(s)	Example
MOD $(X_1, X_2) = X_1 \bmod X_2$	Integral	Integral	$X_1 = 100$; $X_2 = 7$. MOD $(X_1, X_2) = 2$.
MAX $(X_1, ..., X_k)$, $k \geq 2$	Same as arguments	Integral or floating-point	$A = 1$; $B = 14$; $C = 6$. Y = MAX (A, B, C) $Y = 14$.
MIN $(X_1, ..., X_k)$, $k \geq 2$	Same as arguments	Integral or floating-point	$A = 0.1$; $B = 14.0$; $C = 6.1$. Y = MIN (A, B, C) $Y = 0.1$.
SIGN $(X) = \begin{cases} 1, X > 0 \\ 0, X = 0 \\ -1, X < 0 \end{cases}$	Same as argument	Integral or floating-point	If $X = 34$, SIGN $(X) = +1$. If $X = 0$, SIGN $(X) = 0$. If $X = -15$, SIGN $(X) = -1$.
ABS $(X) = \mid X \mid$	Same as argument	Integral or floating-point	If $X = -45.67$, ABS $(X) = 45.67$. If $X = +19$, ABS $(X) = 19$.
PCS (N)	Boolean	Integral or floating-point	If PROGRAM CONTROL SWITCH 3 is ON, PCS(3) = 1. If OFF, PCS(3) = 0.

PROCEDURES

A *procedure* is a closed independent routine which may be executed as a subprogram. This independence makes procedures extremely important features of this compiler. A procedure may be written and checked out independently, a collection of these procedures then being retained as a repository of computing techniques. The flexibility built into the argument structure of procedures allows a specific procedure to be tailored to a variety of situations.

Arguments of Procedures

Whenever a procedure is used, a list of arguments is specified. These arguments may be grouped into three categories: Input arguments, output arguments, and program-reference arguments. Some of these categories may be missing, depending of course upon the specific procedure used.

An input argument may be an expression or an array.

An output argument may be a simple variable, a variable with subscripts, or an array. A program-reference argument may be a statement label, subroutine label, input label, output label, or format label,† any function defined by a FUNCTION declaration, or any other procedure.

It is important to distinguish between an array and an element of an array. A variable with subscripts is an element of an array—it represents a single quantity. An array, however, represents a collection of quantities. When an array is indicated as an argument of a procedure, the procedure is concerned with this entire collection of quantities.

Suppose that the two-dimensional array named M is to be an argument. The notation used for this argument is M(,), the two empty subscript positions indicating that M is two-dimensional. It is also possible to specify that a portion of some array be given to a procedure as an argument. For example, if a procedure requires that a certain argument be a one-dimensional array, the

† These constructs are discussed at length on pages 8-3ff.

array could be chosen as the (I + 1)th row of M(,) by writing M(I + 1,). The single empty subscript position indicates a one-dimensional array. In similar fashion, the (L − 2K)th column of M would be written as M(,L − 2K). In general, the name of an array followed by a subscript list which contains empty subscript positions specifies an array with dimensions equal to the number of empty subscript positions. This holds whether or not some of the subscripts are specified.

To specify a program-reference argument which is to be a label, or the name of a subroutine, input-data set, output-data set, or format list, it is necessary only to write the desired identifier. A function or procedure is specified by writing the name of the function or procedure followed by a pair of parentheses, for example TANH( ).

Note that we now have a distinction between a function and an evaluated function. An evaluated function has its arguments specified; it represents the quantity obtained by applying the definition of the function to those arguments, and is thus an expression. A function, however, represents only the definition.

Functions Used As Arguments

Any function—library, external, or declared—may be used directly as a program-reference argument, with the exception of the intrinsic functions, listed on page 7-2. The exclusion of these intrinsic functions as program-reference arguments was intentional, since the compiler cannot treat this case directly. If a procedure requires a function as one of its arguments, and the user desires to specify it as an intrinsic function, that intrinsic function may be renamed by the use of the FUNCTION declaration. For example, if ABS( ) is to be the function specified, write

```
FUNCTION F(X) = ABS(X)
```

and give the procedure F( ) for its argument.

No provision has been made in the compiler for using a function the arguments of which have been in part specified and in part left empty. This situation causes little inconvenience, however, since the FUNCTION declaration may be used in lieu of such a feature. For example, assume that a function of two arguments Q(X,Y) is available and that it is desired to specify, as an argument to a procedure, that function of one argument Y which is defined by always setting X to (A − B)*3. If the declaration

```
FUNCTION QPRIME(Y) = Q((A − B)*3,Y)
```

is included in the symbolic program, then using the argument QPRIME( ) will produce the desired results.

The entry to a procedure is specified as follows:

GENERAL FORMS:

First form: $\mathcal{P}$ ($\mathcal{IA}$; $\mathcal{OA}$; $\mathcal{PRA}$)
where $\mathcal{P}$ is the name of the procedure being called;
$\mathcal{IA}$ is the list of input arguments;
$\mathcal{OA}$ is the list of output arguments; and
$\mathcal{PRA}$ is the list of program-reference arguments.

Any but not all of the argument lists may be missing, resulting in the following alternative forms:

Second form: $\mathcal{P}$ ($\mathcal{IA}$)
Third form: $\mathcal{P}$ ($\mathcal{IA}$; $\mathcal{OA}$)
Fourth form: $\mathcal{P}$ ($\mathcal{IA}$; ; $\mathcal{PRA}$)
Fifth form: $\mathcal{P}$ (; $\mathcal{OA}$)
Sixth form: $\mathcal{P}$ (; $\mathcal{OA}$; $\mathcal{PRA}$)
Seventh form: $\mathcal{P}$ (; ; $\mathcal{PRA}$)

FUNCTIONS DEFINED BY PROCEDURES

Some procedures define functions—that is, the procedure with a given set of arguments represents a quantity. This is the extended form of the evaluated function mentioned in CHAPTER II. As an example of this sort of construction, suppose a procedure is available to perform definite integration, such as the following:

```
SIMPS (A,B, EPSILON; ; F( ) )
```

where

A and B are the limits of integration;

EPSILON is the maximum tolerable error in the result; and

F( ) is the function to be integrated.

The value associated with the procedure is the value of the definite integral indicated.

Now suppose that we wish to evaluate the equation:

$$J = 4\left[\sqrt{\left(\int_{X-Y}^{X+Y} G_3(T)dT\right)^3}\right].$$

Then the assignment statement

```
J = 4SQRT (SIMPS ( X − Y, X + Y, 1**−6; ; G3( ) )*3 )
```

would suffice.

The Procedure-Call Statement

Rather than evaluating a function, a procedure may be constructed so that it performs a complete operation in itself. Such a procedure is called an 'extension.' All the input-output operations executed by a compiled program are of this kind.

As an example of such a procedure, suppose that a procedure called INVERT has been defined to evaluate the inverse and also to calculate the determinant of a given matrix. The arguments of this procedure are to be:

Input: The order of and the name of the matrix, the determinant and inverse of which are to be evaluated;

Output: The array which is to receive the inverse and the variable which is to receive the computed value of the determinant;

Program-reference: A statement label to which transfer is to be made if the matrix is singular.

The user would then write:

```
INVERT (N, A(,); B(,), D; ERROR4)
```

to set D to the value of the determinant of N × N matrix A(,) and to set B equal to the inverse of the matrix A. A transfer to the statement labeled ERROR4 will occur if A(,) is singular. This procedure call constitutes a complete (operational) statement in itself. Such a statement is called a *procedure-call statement.*

As a second example, suppose that a vector of data points is to be treated by a least-squares smoothing process. Suppose a procedure called SMOOTH is available. The programer might write:

```
SMOOTH ( K, X( ); Y( ) )
```

where the input parameters are first, the number of data points K, and second, the name X( ) of the vector containing them, while the output parameter Y( ) is the name of the vector to receive the computed results.

Machine-Language Procedures

There are three sources by which a procedure may be made available to a compiled program:

First, the procedure may be taken from a magnetic-tape library of machine-language programs which define procedures. These are called *library procedures.*

Second, a machine-language program deck defining a procedure may be included with the cards containing the symbolic program. These are called *external procedures.*

Third, a procedure may be defined in the compiler language. Since library procedures and external procedures are written in the internal language of the BURROUGHS 220, any features of the computer for which no direct provision is made in the compiler language may be made available to a compiled program—for example, input-output equipment, multiple-precision arithmetic, detection of overflow, etc. In addition, the library contains a selection of commonly used procedures to evaluate the elementary functions. While many of these functions may be expressed in the compiler language, the convenience of having these procedures 'on call' without the programer directly providing their definitions makes their inclusion in the library worthwhile.

APPENDIX E describes the preparation of library and external procedures; APPENDIX F describes the library procedures currently available.

Declaration of Procedures

Procedures may be made available to the compiled program by means of the PROCEDURE declaration.

GENERAL FORM:

PROCEDURE $\mathcal{P}$ ($\sim$ $\mathcal{LP}$ $\sim$);
  BEGIN $\mathcal{S}_1$; $\mathcal{S}_2$; $\sim$; $\mathcal{S}_n$; END

where $\mathcal{P}$ is an identifier which names the procedure being declared; the $\mathcal{S}$'s are the statements and declarations making up the definition of the procedure; and $\mathcal{LP}$ is the list of parameters to be used by the procedure. As an alternative form, END may be followed by the name of the procedure and then by a pair of parentheses.

The List of Parameters

The *list of parameters* of the procedure declaration consists of identifiers and punctuation marks, these identifiers serving as names of the various parameters.

Input parameters representing input arguments which are variables or expressions must be simply identifiers. Input parameters which represent arrays must be identifiers followed by pairs of parentheses, perhaps containing commas. The number of empty subscript positions within the pair of parentheses specifies the number of dimensions of the array. Output parameters have the same form as input parameters and represent output variables or output arrays. Program-reference parameters which may be used include identifiers representing labels (statement, subroutine, segment, input, output, or format labels) or identifiers, each followed by a pair of parentheses representing functions or other procedures.

The list of parameters of a PROCEDURE declaration takes the same general form as the list of arguments of a procedure call. Let $\mathcal{IP}$, $\mathcal{OP}$, and $\mathcal{PRP}$ represent input, output, and program-reference parameters respectively. Then the list of parameters may assume the following configurations:

First: ($\mathcal{IP}$; $\mathcal{OP}$; $\mathcal{PRP}$)
Second: ($\mathcal{IP}$)
Third: ($\mathcal{IP}$; $\mathcal{OP}$)
Fourth: ($\mathcal{IP}$; ; $\mathcal{PRP}$)
Fifth: (; $\mathcal{OP}$)
Sixth: (; $\mathcal{OP}$; $\mathcal{PRP}$)
Seventh: (; ; $\mathcal{PRP}$)

Independence of Declared Procedures

The compound statement defining a procedure is written in terms of the identifiers appearing in the list of parameters of the PROCEDURE declaration, together with any other identifiers required.

The definition of a procedure should be considered as a symbolic program which is independent of the program in which the declaration occurs; that is, all identifiers appearing within a PROCEDURE declaration are defined only in terms of the declaration itself. Identifiers spelled identically both inside and outside of any particular PROCEDURE declaration are in no way associated. There is one exception to this rule: *After a procedure is declared, the identifier which names it is recognized as such throughout the subsequent program except where it is used as a dummy parameter for another declaration of a procedure or function.* Indeed, the definition of a procedure might be construed as adding another feature to the compiler language, since the name of a procedure is recognized throughout the program just as are the reserved words FOR, GEQ, etc.

Declarations with Procedures

The procedure definition must contain a sufficient number of declarations to describe the identifiers appearing either as parameters or in any other form within that definition. Parameters which represent arrays, functions, or other procedures are identified as such by the punctuation associated with them in the list of parameters. For example, if W(, ,) appears as an input or output parameter, W should not appear in an ARRAY declaration within the PROCEDURE declaration. However, any parameters which represent quantities within the PROCEDURE declaration must have their types specified, either explicitly within a type declaration, by prefixes, or by default. (See CHAPTER V.) These declarations have no force outside the PROCEDURE declaration. A parameter representing a label of one sort or another is identified as such by its inclusion in the program-reference portion of the list of parameters without a trailing pair of parentheses, this constituting a sufficient identification for labels.

Parameters of Value and Name

It is necessary to distinguish two classes of parameters, *parameters of value* and *parameters of name.*

Parameters of value are variables within the procedure. These variables will be set to the values of their corresponding arguments whenever the procedure is called. Any change in them occurring within the procedure (for example, appearing as the left-hand member of an assignment statement) will have no effect on the variable or variables which make up the corresponding argument.

On the other hand, *parameters of name* are associated with their corresponding arguments in all respects. For example, if PAR is a parameter of name for a certain PROCEDURE declaration and, for some call of that procedure, ARG is the corresponding argument, then the effect is exactly as though the identifier ARG were substituted for the identifier PAR throughout the PROCEDURE declaration.

Input variables (or expressions) are parameters of value. Input arrays, all output parameters, and all program-reference parameters are parameters of name. (There is thus no real distinction between an array indicated as an input or as an output parameter.)

Construction of Procedures

As mentioned earlier, a procedure is an independent program complete with its own declarations and statements. This program is contained within the BEGIN...END pair noted in the general form. The program defining the procedure is entered at the first statement following the BEGIN and continues in accordance with the sequence specified. As with the SUBROUTINE declaration, a RETURN statement must be included at each exit point of the procedure to return control to the point directly following the procedure call.

A procedure which is to serve as a function must include a *procedure-assignment statement.*

GENERAL FORM:

$\mathcal{P}\ (\ ) = \mathcal{E}$

where $\mathcal{P}$ is the name of the procedure being declared and $\mathcal{E}$ is an expression. The effect of this statement is to assign the value of $\mathcal{E}$ to the procedure. Immediately after this statement is executed, a RETURN statement must be executed.

Whenever a procedure-assignment statement is used, the type of the procedure is determined from the declarations of type within the PROCEDURE declaration.

FUNCTION and SUBROUTINE declarations may appear within a PROCEDURE declaration. *However, one* PROCEDURE *declaration may not appear within another procedure.*

If a SUBROUTINE declaration appears within a PROCEDURE declaration, a RETURN statement within the subroutine causes an exit from the subroutine, not from the procedure.

Procedures must be declared prior to their first use.

Examples of PROCEDURE Declarations

As a first example, we shall construct a procedure to perform linear interpolation of a tabular function of one variable, V. The parameters of the procedure will be two vectors, X ( ) and Y ( ), representing the independent and dependent variables, a value of the independent variable, an integer N representing the number of entries in the table, and finally a statement label, RANGE, to which transfer is made in the event that $V<X(1)$ or $V\geq X(N)$. This procedure is to be used as a function, the value of which is the result of the interpolation.

EXAMPLE:

```
PROCEDURE INTERP (X( ), Y( ), V, N; ; RANGE); BEGIN
  INTEGER I, N;
  IF (V LSS X(1) ) OR (V GTR X(N) ); GO TO RANGE;
  I = 1; UNTIL V LEQ X(I); I = I + 1;
  INTERP( ) = Y(I − 1) + (Y(I − 1) − Y(I) ) (V − X(I − 1))/
  (X(I − 1) − X(I) ); RETURN END INTERP( )
```

As another example, we shall construct the integration procedure using Simpson's Rule mentioned earlier in this chapter.

```
PROCEDURE SIMPS (A,B,EPSILON; ; F( ) ); BEGIN
K = L = F(A) + F(B); H = B − A;
GO TO ITER; UNTIL ABS( (K − 2M)/K)  LSS EPSILON;
BEGIN ITER..Q = H/2; S = 0; M = K;
FOR X = (A + Q,H,B) ; S = S + F(X);
K = L + 4S ; L = (L + 2S)/2; H = Q END;
SIMPS( ) = K.Q/3; RETURN END SIMPS( )
```

A procedure for multiplication of square matrices:

```
PROCEDURE MATRIMULT (N, A(,), B(,); C(,) ); BEGIN
  INTEGER I,J,K,N;
  FOR I = (1,1,N); FOR J = (1,1,N); BEGIN
    S = 0; FOR K = (1,1,N); S = S + A(I,K).B(K,J);
    C(I,J) = S END; RETURN END
```

(Note, in calling this procedure, that the matrix C must be different from A and B.)

The following procedure solves a set of n equations in n unknowns.

```
COMMENT SOLVE EQUATIONS WITH SELECTION
  OF BEST PIVOTAL ROW;
PROCEDURE JORDAN (N,A(,); X( ));
  BEGIN INTEGER I,J,K,L,N;
  FOR K = (N + 1, −1, 1); BEGIN D = 0;
    FOR I = (2,1,K); IF ABS (A(I − 1,1) ) GTR D;
      BEGIN L = I − 1; D = ABS (A(L, 1) ) END;
    IF L − 1 NEQ 0;
  FOR J = (1,1,K); BEGIN D = A(L,J);
    A(L,J) = A(1,J); A(1,J) = D END;
  FOR I = (1,1,N); X(I) = A(I,1);
  FOR J = (2,1,K); BEGIN D = A(1,J)/X(1);
    FOR I = (2,1,N); A(I − 1, J − 1) = A(I,J) − X(I).D;
    A(N,J−1) = D END END; RETURN END JORDAN( )
```

External Declaration

By the use of this declaration, a programer may define a statement or a procedure in terms of machine language and include it in a compiled program.

GENERAL FORMS:

First form:

EXTERNAL STATEMENT £

where £ is the label of an external statement.

The first form declares the program represented by £ to be a statement which will behave similarly to any other active statement in the language. The label of this statement will be £.

Second form:

EXTERNAL PROCEDURE £ ($\mathcal{P}_1$, ⋯, $\mathcal{P}_n$); { INTEGER / BOOLEAN / FLOATING / REAL } £

where £ is the label of the external procedure, and $\mathcal{P}_1$, ⋯, $\mathcal{P}_n$ is a list of parameters.

Both forms tell the compiler that a machine-language program to define £ follows after the FINISH card (See APPENDIX E).

The second form defines the program represented by £ to be a procedure which will behave like any other procedure in the language. Note that if the procedure is to define a function, then its declaration of type must follow it immediately.

EXAMPLES:

First form:

```
EITHER IF V GTR NMAX; BEGIN EXTERNAL STATEMENT
ERROR; GO RESET END; OR IF K LSS EPS; GO ERROR END
```

Second form:

```
EXTERNAL PROCEDURE COMPLEXMULT (A, B, C, D; X, Y);
FLOATING COMPLEXMULT; SREAL = 0; SIMAG = 0;
FOR I = (1,1,10); BEGIN COMPLEXMULT (AREAL(I),
AIMAG(I), BREAL(I), BIMAG(I); TREAL, TIMAG);
SREAL = SREAL + TREAL; SIMAG = SIMAG + TIMAG END
EXTERNAL PROCEDURE SPERR(;;FRDEC);
IF V GTR NMAX; SPERR(;;NERR)
```

```
PROCEDURE MATRIMULT (M, N, P, A( ), B( ), ROW, COLUMN,
OUTTAPE; C( )); BEGIN INTEGER OTHERWISE;
FLOATING A, B, C; EXTERNAL PROCEDURE REWIND (U);
EXTERNAL PROCEDURE MAGREAD (N, U, A( ));
EXTERNAL PROCEDURE MAGWRITE (N, U, A( ));
REWIND (ROW); REWIND (COLUMN); REWIND (OUTTAPE);
FOR I = (1, 1, M); BEGIN MAGREAD (P, ROW, A( ));
FOR J = (1, 1,N); BEGIN MAGREAD (P, COLUMN, B( ));
BEGIN S = 0; FOR K = (1, 1, P); S = S + A(K) B(K);
C (J) = S; REWIND (COLUMN) END;
MAGWRITE (N, OUTTAPE, C())  END; RETURN END
```

INPUT
PREPARATION OF DATA CARDS
OUTPUT
FORMAT
EDITING

VIII . . .

input-output techniques

THIS CHAPTER DISCUSSES those features of the compiler relating to communication of information between the computer and the input-output equipment. In general, the compiler input-output operations are accomplished by means of machine-language procedures (either external procedures or library procedures). Three declarations are provided to aid in the input-output processes—the INPUT, OUTPUT, and FORMAT declarations.

INPUT OF INFORMATION

The INPUT Declaration

The INPUT declaration associates with identifiers the ordered sets of numbers which are to be read into the computer as units. These sets of numbers will be called *input-data sets* and the identifiers are termed the *input data-set labels.*

GENERAL FORM:

INPUT ($\mathcal{I}(\mathfrak{DS})$, ⋯, $\mathcal{I}(\mathfrak{DS})$)

where each $\mathcal{I}$ is the identifier declared to be the name of the corresponding input-data set $\mathfrak{DS}$. As explained previously (page 5-1) the parentheses around the list of input data-set labels in the above declaration are included at the programer's option. Input-data sets consist of a set of list elements separated by commas. The simplest of these sets is merely a list of variables.

First form:

$\mathcal{V}_1$, $\mathcal{V}_2$, ⋯, $\mathcal{V}_n$

where each $\mathcal{V}$ may be either a simple variable or a variable with subscripts.

EXAMPLE:

```
INPUT DATA1 (X,Y,Z), DATA2 (P(I), Q(I))
```

This declaration defines DATA1 to be the set of variables X, Y, and Z (in that order) and DATA2 to be the set P(I) and Q(I), using the current value of I for the subscript.

The next list element to be considered is termed the *iterated variable.*

Second form:

FOR $\mathcal{V}_1 = \mathcal{IL}$; ⋯; FOR $\mathcal{V}_k = \mathcal{IL}$; $\mathcal{V}$

where FOR $\mathcal{V}_1 = \mathcal{IL}$ through FOR $\mathcal{V}_k = \mathcal{IL}$ are FOR clauses which control the iteration of the variable (with subscripts) $\mathcal{V}$. One or more of these FOR clauses may precede $\mathcal{V}$.

EXAMPLE:

```
INPUT ( VECTOR(FOR L = (1,1,F); Z(L) ),
MATRIX ( FOR I = (1,1,N); FOR J = (1,1,M); E(I,J) ) )
```

This declaration defines VECTOR to be the name of the input-data set consisting of the variables

Z(1), Z(2), Z(3), ⋯, Z(F − 1), Z(F),

and MATRIX to be the name of the input-data set consisting of the variables

E(1,1), E(1,2), ⋯, E(1,M), E(2,1), ⋯, E(N,M).

Note that the values for F, M, and N must have been assigned elsewhere in the program.

The remaining forms of the input-data set consist of combinations of the first and second forms. In the first form any $\mathcal{V}$ may be replaced by any input-data set and still be a valid data set. In the second form, the $\mathcal{V}$ may

be replaced by any data set enclosed in parentheses and still be a valid data set. In this case, all of the enclosed data set is iterated. These parentheses may be replaced by BEGIN and END if desired.

Third form:

𝔇𝔖, 𝔇𝔖, ⋯, 𝔇𝔖

Fourth form:

FOR $\mathfrak{V}_1 = \mathfrak{gL}$; ⋯; FOR $\mathfrak{V}_k = \mathfrak{gL}$; (𝔇𝔖)

The data-set lists given by the third and fourth forms may of course be used as data-set lists within their own definitions, thus providing for a very high degree of flexibility.

EXAMPLES:

```
INPUT EQUATIONS (N, FOR I = (1,1,N);
  (FOR J = (1,1,N); M(I,J), C(I) ) )
```

This data set consists of the variables N, M(1,1), M(1,2), ..., M(1,N), C(1), M(2,1), ..., M(2,N); C(2), M(3,1), ..., M(N,N), and C(N).

Note that as soon as a variable (in this case N) has been read into the computer, it is immediately available for use in a FOR clause or within a subscript expression.

Input Procedures

A machine-language procedure is employed to obtain numbers from the input medium. The name of an input data-set list usually will appear as a program-reference parameter of an input procedure.

A READ procedure used very frequently is given below.

There is an important feature of all input procedures which should be noted, so that caution may be exercised in their use. *Any* FOR *clause with an* INPUT *declaration affects the value of the variable being stepped in exactly the same manner as a* FOR *clause controlling a statement.* Thus if an input procedure is contained within the scope of a FOR clause and the INPUT declaration to which it refers contains a FOR clause, *the variables stepped by the two iterations must be different.* This also applies to the output procedures described later in this chapter.

The READ Procedure

The READ procedure provides card input for compiled programs. It may be called in either of two ways:

GENERAL FORMS:

First form:

READ (; ; 𝔤𝔇𝔏)

Second form:

READ (; 𝔰; 𝔤𝔇𝔏)

where 𝔤𝔇𝔏 represents the name of some input data-set list and 𝔰 is a Boolean variable.

The effect of this procedure is to read a card and scan it for information. As numbers are found, they are placed in storage according to the specifications of 𝔤𝔇𝔏. Sufficient cards are read to supply all the numbers requested by 𝔤𝔇𝔏; if the entries on the last card read are not requested, they are lost. It is possible to mark a card as being a *sentinel card.* If the second form was used, the Boolean variable 𝔰 is set to one when a sentinel card is found and the input process is terminated; 𝔰 is otherwise set to zero. Using the first form will cause the word SENTINEL to be ignored.

Preparation of Data Cards

Data cards are punched with a digit five in column 1; the remainder of the card is available for data. An integer is punched as a string of no more than ten contiguous digits, preceded by an optional + or −. Floating-point numbers are punched as a string of digits containing a decimal point.†

No more than eight significant digits may appear in a floating-point number. Leading zeros are not significant; trailing zeros are. A sign may precede the number. A scale factor (power of ten) may be attached to a floating-point number by following the number with a ',' (not '**'), an optional sign, and a two-digit integer.

At least one blank column must separate the numbers on a card. A number may not be broken between successive cards.

Alphanumeric input information is punched on a card bracketed with semicolons; consequently the character ';' may not be used within the alphanumeric information itself. For example, if columns 13 through 41 of a card are to be entered alphanumerically (six words) then a ';' should be punched in columns 12 and 42.

Alphanumeric information, in the form of integers, may be stored in the computer. One integer, ten digits in length, is stored in the computer for every five consecutive characters of alphanumeric information. The internal representation is two digits per alphanumeric character as described in APPENDIX C of *Operational Characteristics of the* BURROUGHS 220 *Electronic Data Processing System.* If the number of columns of alphanumeric information is not a multiple of five, the last integer formed will have zeros appended. The characters read from the card will be justified left in this word.

An asterisk will cause all information to its right on the card to be ignored (if the asterisk is not within an alphanumeric string).

† As opposed to the representation used for constants in the symbolic program, the decimal point may also appear on data cards at either end of the string.

EXAMPLES:

1 37 −4724 + 0394	are integers
3.0 −.9 + 3.1416 8.	are floating-point numbers
−2.99,9 32.4,−13 + 7.2,+2	are floating-point numbers with scale factor

; THE GOAT OF HOGAN ; is an alphanumeric entry

A sentinel card is identified by punching the word SENTINEL starting at column 2 and followed by a blank column:

COLUMNS	ENTRY
1	5
2- 9	SENTINEL
10	
11-	*(optional)*

OUTPUT OF INFORMATION

The OUTPUT Declaration

The OUTPUT declaration associates with identifiers the ordered sets of expressions which are to be written out of the computer as groups. These sets of expressions are called *output-data sets* and the identifiers are termed *output data-set labels.*

GENERAL FORM:

OUTPUT ($\mathfrak{g}$ ($\mathfrak{DS}$), ⋯, $\mathfrak{g}$ ($\mathfrak{DS}$))

where each $\mathfrak{g}$ is the identifier declared to be the name of the corresponding output-data set $\mathfrak{DS}$. (See page 5-2 for the explanation of the use of the outside parentheses.) Output-data sets are constructed in a manner precisely analogous to input-data sets *with the following exception: Although only values of variables may be read into the computer, the value of any expression may be written out. Thus wherever a variable has been specified in the description of an input-data set, the reader may substitute an expression in an output-data set.*

EXAMPLE:

OUTPUT RESULTS ((R − 1) (S − 1), N(R), N(S), 8.4)

Output Procedures

A machine-language procedure is employed to transmit numbers to the output medium. The name of an output-data set appears as the program-reference parameter of an output procedure.

The WRITE Procedure

The WRITE procedure provides for writing the edited output of a compiled program on the LINE PRINTER. the CARD PUNCH, or the SUPERVISORY PRINTER.

First form:

WRITE (; ; $\mathfrak{DS}$, $\mathfrak{FL}_1$)

Second form:

WRITE (; ; $\mathfrak{FL}_2$)

where $\mathfrak{DS}$ is the name of an output-data set and $\mathfrak{FL}_1$ and $\mathfrak{FL}_2$ are the names of format strings.

The effect of the first form is to produce as an output the numbers specified by the output-data set $\mathfrak{DS}$ in accordance with the format specified by $\mathfrak{FL}_1$. The second form is used to write the headings indicated by the format string $\mathfrak{FL}_2$.

The format strings mentioned are described in the next section.

CONSTRUCTION OF FORMATS

The FORMAT Declaration

The FORMAT declaration has been designed specifically for use with the WRITE procedure, and serves to attach names to strings of characters called *formats*, the latter describing the appearance of the output page or card. Such a name is then used as a parameter by the WRITE procedure.

GENERAL FORM:

FORMAT ($\mathfrak{g}$($\mathfrak{FS}$), $\mathfrak{g}$($\mathfrak{FS}$), ⋯, $\mathfrak{g}$($\mathfrak{FS}$))

where the $\mathfrak{FS}$'s are formats and the $\mathfrak{g}$'s are the names of those formats;† (outside parentheses are again optional).

The format consists of a sequence of phrases which are separated by commas and perhaps grouped by parentheses. These phrases occur in two forms:

First form:

rLw.d

Second form:

$\mathfrak{AS}$

† These formats are not to be confused with the format bands used in the BURROUGHS CARDATRON®.

In the first form, the symbols *r*, *w*, and *d* represent integers and *L* represents a letter. The integers *r* or *d* or both are sometimes omitted, reducing the first form to *Lw.d*, *rLw*, or *Lw* in these cases.

The integer *r* specifies the number of times a phrase is to be repeated. If *r* is omitted, the phrase is executed once. Phrases of the first form are divided into two classes—editing phrases and activation phrases.

In the second form of a phrase, $\mathfrak{as}$ represents any sequence of characters which does not contain an '*'. This phrase is used for placing alphanumeric titles or other indicative information in the output line. It is called the alphanumeric-insertion phrase.

Repeat Phrases

A list of phrases may be placed in parentheses to constitute a compound phrase. These parentheses may be nested to any depth.

Definite-Repeat Phrase:
$r(\mathfrak{fs})$

Indefinite-Repeat Phrase:
$(\mathfrak{fs})$

where $\mathfrak{fs}$ is a format string. The definite-repeat phrase uses the format string *r* times in succession; the indefinite-repeat phrase uses the format string repeatedly until there are no more variables to print.

An entire format string is treated as if it were enclosed in parentheses specifying indefinite repeat. The interpretation of a format is terminated when there are no more variables to print and the right parenthesis of an indefinite repeat is encountered. If there are fewer variables to print than are called for, the I, X, F, S, and A phrases are interpreted as blank-insertion phrases, B*w*, to fill in the remaining spaces.

EXAMPLES:

```
3(5F15.8,W0)
*TITLE*,W3,(5I11,W0)
```

The first example is equivalent to:

```
5F15.8,W0,5F15.8,W0,5F15.8,W0
```

The second example will print a line which reads

```
TITLE
```

and will then print a series of lines, each line consisting of five integer variables, until no more variables are available from the OUTPUT declaration. If the number of variables to be printed is not a multiple of five, a sufficient number of blanks will be inserted to finish out the line.

Editing Phrases

A numeric editing phrase specifies how a number is to be edited. There are six such phrases:

I*w* The I phrase specifies that an integer is to be printed (or punched) in a field *w* columns wide. The integer will be normalized right in that field and will have its leading zeros suppressed. If the integer being edited is negative, a '−' (minus sign) precedes it. The value of *w* must be sufficiently large to accommodate the largest integer to be encountered together with any possible minus sign.

X*w.d* The X phrase specifies that a floating-point number is to be truncated to *d* places following the decimal point and printed in a field *w* columns wide. The value *w* must be sufficiently large to accommodate the largest number to be printed along with its decimal point and any possible minus sign.

F*w.d* The F phrase specifies that a floating-point number is to be truncated to *d* significant digits and printed (or punched) in floating-point form as follows:

A '−' (if the number is negative), a '.', *d* digits, a ',', a '−' (if the power of 10 associated with the number is negative), and two digits representing that power of 10. Thus to accommodate negative numbers $w \geq d + 6$.

S*w.d* The S phrase specifies that a floating-point number is to be truncated to *d* digits and printed (or punched) in a field *w* columns wide. A decimal point will be inserted at the appropriate position to cause the printing (or punching) of *d* significant digits if possible. If a number is less than 0.1, zeros will be inserted between the decimal point (which will appear at the extreme left) and any significant digits printed, punched, or typed.

A*w* The A phrase allows integers to be printed (or punched) in their alphanumeric equivalents. A single A phrase will produce, as an output, *w* characters, five characters of alphanumeric information being translated from each ten-digit integer. If *w* is not a multiple of five, the least significant portion of the last integer will be ignored.

If in any case the field width *w* which has been specified is less than the width required by the output information, or if there are undefined conditions in the format specifications, an asterisk will be printed in the corresponding field.

B*w* The phrase B*w* will cause *w* blank columns to be inserted in the edited line.

EXAMPLES:

We list here several phrases and a typical result of the editing they specify. In each example the first of the two lines indicates the result of the editing process, where the symbol # indicates an editing space. The line directly below each example shows the equivalent line as printed.

PHRASE	RESULT
4I7	#####13#−72431####342######0
	13 −72431 342 0
3X5.2	##.13#−.52#1.74
	.13 −.52 1.74
X12.10	#.0000000015
	.0000000015
F10.3	##.472,−03
	.472,−03
3F12.4	##−.3942, 07###.4311,−03###.0000, 00
	−.3942, 07 .4311,−03 .0000, 00
5S9.5	###3427.1##−32.993##−.10206###13788.###.00014
	3427.1 −32.993 −.10206 13788. .00014

A12 The use of this phrase, in conjunction with
4856596245434845626345590057 41
will result in the printout
HORSECHESTER

Alphanumeric Insertion Phrase

The phrase *αs* inserts the characters comprising the string αs into the line being edited.

EXAMPLES:

PIPE DIAMETER
TRANSCONDUCTANCE-MICROMHOS
HIGH - LOW - CLOSE - NET CHG.
DURCHSCHLAGFESTIGKEIT - VOLT

Activation Phrases

An *activation phrase* specifies that the line described by the preceding phrases is to be sent to the output device.

Whenever a line is written out in this manner, the image in memory of the line is reset to blanks. Thus, if an activation phrase is repeated, only the first execution produces any printed results; the repeat merely provides vertical spacing. Four activation phrases are provided:

W*w* The W phrase provides for output on the LINE PRINTER. The value of *w* specifies the 'c-digit' to be used for printer control. If *w* is omitted, it is assumed to be zero. If the control panel is wired as specified in BURROUGHS CORPORATION TECHNICAL BULLETIN No. 17, *Control Panel Wiring for Type* 407 *with* BURROUGHS 205 *or* 220 CARDATRON®, *w* is interpreted as follows:

w	RESULT
0	Single space before printing
1	Eject page after printing
2	Single space before and after printing
3	Eject page before printing
4	Double space before printing
5	Skip to channel 2 before printing
6	Double space before printing, single space after printing
7	Skip to channel 3 before printing

P The P phrase specifies that the edited line is to be punched into a card. Note that, with the exception of the A phrase, the edited form of the numbers is compatible with the requirements for input-data cards. Thus the output cards produced by the WRITE procedure may be used for input to the computer under control of a READ procedure.

C*w* The action of the C phrase combines those of the W and P phrases. Only the first 80 columns of the edited line will be punched.

T*w* The T phrase specifies that the edited line is to be typed on the SUPERVISORY PRINTER. The value of *w* specifies the number of carriage returns to be executed prior to typing.

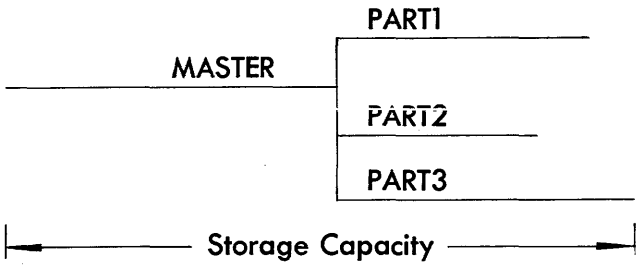
SEGMENTATION
OVERLAYS

IX . . .

overlay techniques

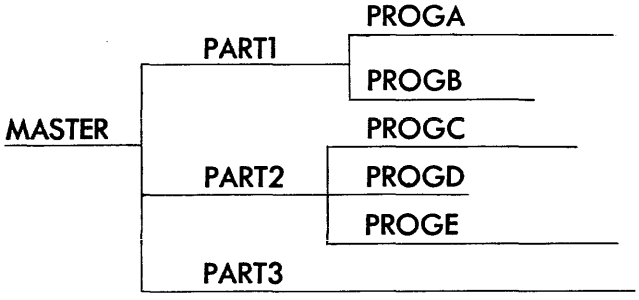
THE USE OF STORAGE OVERLAYS, though not essential for a majority of problems, is required whenever a program is too large for the amount of available storage. In such a case the program is coded in segments; these segments are then called into memory as a sequence of overlays under control of a master routine.

To illustrate this graphically:



Here MASTER is the name of the master routine and PART1, PART2, and PART3 are the names of the segments. The lengths of the horizontal lines may be taken as proportional to the number of instructions in the master routine and in the various segments. It will be noticed that in this case only sufficient storage capaccity for MASTER and PART 3 is required.

Each of the segments may in turn have subsegments. All of these segments then become *the master routines for the control of their respective subsegments.* For example:



The master for PART1, PART2, and PART3 thus is MASTER; the master for PROGC, PROGD, and PROGE is PART2, etc.

THE SEGMENT DECLARATION

The SEGMENT declaration is used to indicate the division of the program into segments.

GENERAL FORM:

SEGMENT $\mathcal{I}$; BEGIN S_1; S_2; ⋯; S_n END

where $\mathcal{I}$ is an identifier which serves as the name of the segment (segment label) and S_1 through S_n are the statements which make up the segment. If desired the segment label may follow the word END as is the case with any other compound statement.

For example, the segmentation diagramed above would be written as:

```
MASTER..⋯⋯⋯⋯⋯;
  SEGMENT PART1; BEGIN⋯⋯⋯⋯⋯;
    SEGMENT PROGA; BEGIN⋯⋯END PROGA;
    SEGMENT PROGB; BEGIN⋯⋯END PROGB
      END PART1;

  SEGMENT PART2; BEGIN⋯⋯⋯⋯⋯;
    SEGMENT PROGC; BEGIN⋯⋯END PROGC;
    SEGMENT PROGD; BEGIN⋯⋯END PROGD;
    SEGMENT PROGE; BEGIN⋯⋯END PROGE
      END PART2;

  SEGMENT PART3; BEGIN⋯⋯END PART3
```

THE OVERLAY STATEMENT

The OVERLAY statement is used to call a segment of the compiled program into storage.

GENERAL FORM:

OVERLAY *ℊ*

where *ℊ* is the segment label of the segment to be called. There are three rules governing the use of the OVERLAY statement:

First: Any segment may be overlaid only by a segment of the same immediate master.

Second: The OVERLAY statement which calls a segment into storage must appear in the master routine corresponding to the segment being called. Otherwise, segments may be called in as many times as the user wishes, in any order.

Third: The OVERLAY statement does not by itself produce a transfer to the segment being called. Such a transfer must be provided for separately by the user. *The segment label may not be used as a statement label for effecting this transfer.*

EXAMPLE:

```
BOOLEAN S; REAL A, B; INTEGER OTHERWISE;
ARRAY A(200);
RD..READ(;S;INDAT); IF S; BEGIN OVERLAY SENTINEL;
N = N + 1; GO TO START END; N = N + 1; OVERLAY ZERO;
GO STARTP;
INPUT INDAT (B, C, FOR I = (1, 1, 200); A(I) );
FORMAT FR (I5, 5F20.8, W0);

SEGMENT ZERO; BEGIN STARTP..
FOR I = (1, 10, 200); WRITE (;; FR, OUTP); GO TO RD;
OUTPUT OUTP (I, FOR J = (I, 2, I + 8);
SIN (COS(LOG(A(J))))) END ZERO;

SEGMENT SENTINEL; START..BEGIN FOR I = (1, 10, N – 10);
WRITE (;; FR, OUT); WRITE (;; FRR, OUTR);
STOP 0757007250; GO RD;
OUTPUT OUT(I, FOR J = (I, 2, I + 8); A(J));
OUTR(I, FOR J = (I, S, I + MOD(N, 10) – 2); A(J)),
FORMAT FRR (*I = *, I4, 5(*G(I) = *, F14.6), W0, W3)
  END SENTINEL; FINISH;
```

ERROR MESSAGES
MONITORING
MEMORY PRINTOUT

X . . .

diagnostic facilities

THE COMPILER has been equipped with a set of diagnostic aids. Some of these are operative at the time of compilation, others at the time the object program is run. These aids take the form of error messages concerning the symbolic program, a symbolic snapshot feature, a symbolic memory printout, and messages on the LINE PRINTER produced by machine-language procedures when unsuitable arguments are presented to them. A listing of the compiled program may also be obtained as a desperation tactic.

ERROR MESSAGES DURING COMPILATION

While compilation is in progress, a copy of the original symbolic program is written on the LINE PRINTER. If the compiler detects an error in the program, a message is also written on the printer. Whenever an error is detected, the compiler attempts to continue processing the symbolic program. Since the compilation technique of necessity depends on the syntax of the symbolic program for its classification of identifiers and statements, an error in syntax may well produce a misclassification. If the remainder of the symbolic program is processed with the compiler in such a 'confused' state about the characteristics of the program, a proliferation of error messages may result.

If things get altogether out of hand, the compiler may even come to a disorderly stop. Although such generation of spurious error messages is unfortunate, it has been considered preferable to attempt to continue processing, in order to discover as many errors as possible in a single pass rather than to force the compiler to abandon the compilation after the first error is detected.

The programer himself must make the distinction between genuine errors and those introduced by previous errors. However, it should not be assumed that all errors in syntax will produce error messages.

No usable compiler can provide error messages which explicitly describe every possible syntactical error. However, sufficient information usually is gained from the error message to permit straightforward correction of the program.

The following is a partial list of error messages and some suggestions as to their possible cause.

COMPILER CAPACITY EXCEEDED

The internal storage capacity of the compiler has been exhausted. The compiler stops with rC = 0000007777.

CONSTANT OUT OF RANGE

A constant too large for the internal representation of the BURROUGHS 220 was either written in the symbolic program or was formed when the compiler combined constants arithmetically.

DUPLICATE PROCEDURE NAME

The identifier assigned as the name of a procedure by a PROCEDURE declaration (or the name of a function by a FUNCTION declaration) has appeared in another context in the program.

DUPLICATE STATEMENT LABEL

An identifier has been employed in another context in the program. (Note that this may be the result of calling a function or procedure before it has been declared.)

EXTRA OPERAND

This error message may arise from a wide variety of causes. Check for omitted commas, semicolons, or other punctuation; for proper spelling of reserved identifiers, and for a space imbedded within an identifier; or for the omission of a space between contiguous identifiers.

EXTERNAL PROGRAM NOT DECLARED

The external procedure or external statement declared on some name card was not declared within the symbolic program.

EXTRA LEFT PARENTHESIS

This error message may occur only at the end of processing the symbolic program. It is usually caused by the inclusion of a spurious 'BEGIN' or '(' or by the omission of a required 'END' or ')'. Errors in the syntax of FOR or alternative statements, or in any of the declarations, may also introduce unwanted left parentheses into the program even though the programer did not explicitly write them.

EXTRA RIGHT PARENTHESIS

This error is usually caused by the inclusion of a spurious 'END' or ')' or by the omission of a required 'BEGIN' or '('. As with the extra left parenthesis, check on the syntax of control statements and declarations. The compiler tries to recover from this error by introducing a left parenthesis to match the right parenthesis in question. If the right parenthesis was merely misplaced, this will produce the message EXTRA LEFT PARENTHESIS at the end of compilation.

IMPROPER ARGUMENT OF MOD FUNCTION

The arguments of the intrinsic function MOD must be of integer type.

IMPROPER ARRAY DECLARATION

An error in syntax has occurred in an ARRAY declaration. Check in particular to see that all of the dimensions are specified as integer constants.

IMPROPER ASSIGNMENT STATEMENT

An expression or a constant has occurred to the left of an '='.

IMPROPER BOOLEAN OPERAND

An attempt has been made to use a floating-point quantity as a Boolean operand. The compiler cannot detect integer quantities used as Boolean operands.

IMPROPER CHARACTER PAIR

Two successive characters (omitting spaces) have occurred which are meaningless, for example, '+)', '=*', or ',;'. The compiler will ignore the second of these characters and continue scanning.

IMPROPER CHECK SUM

A failure has occurred in the magnetic-tape system. The compiler will reread the bad tape block repeatedly in an attempt to bring in the information correctly.

IMPROPER EMPTY SUBSCRIPT POSITION

One of the character pairs '(,' ',,' ',)' or '( )' has occurred in the wrong context.

IMPROPER FUNCTION ARGUMENT

An expression has been written as an output or program-reference argument; a label, function name, or procedure name has been used as an input or output argument; or an array has been used as a program-reference argument in a procedure call.

IMPROPER INPUT DECLARATION

An expression or constant has been used as a quantity to be read in.

IMPROPER RELATION OPERATOR

One of the arithmetic relational operators has occurred in the wrong context.

IMPROPER SCALE FACTOR

An attempt has been made to use a scale factor which is not an integer constant.

IMPROPER STATEMENT LABEL

An identifier has been used as a label, in conflict with previous context.

IMPROPER SUBSCRIPT

Either too many or too few subscripts have been associated with an array.

IMPROPER VARIABLE SYMBOL

An identifier has been used as a variable, in conflict with previous context which implied that it was not a variable.

MISPLACED ARITHMETIC OPERATION

One of the symbols + − · / or * has been used in the wrong context.

MISPLACED DECIMAL POINT

The symbol . has been used in the wrong context.

MISSING OPERAND

A misplaced operand may result in the error messages MISSING OPERAND and EXTRA OPERAND appearing in the printout of the compilation. The compiler expected an operand and could not find it. Check the formation of identifiers and the syntax of control statements.

MISSING NAME CARD

The first card of either an external procedure or an external statement was not a name card.

UNDEFINED LABEL –£

The label £ was not defined within the symbolic program. Only the first five characters of £ will be printed if it was an identifier. An integer is printed in its entirety.

PREFIX PROCEDURE NOT DEFINED

The procedure used as a prefix on an equivalence card was not declared within the symbolic program.

MISSING FINISH PSEUDO-OP

The machine-language deck of an external program was not terminated by a pseudo-operation for FINISH.

MISSING FINISH CARD

The FINISH card required after the last machine-language deck was not present.

MISSING FIELD ON HEADER CARD

A name card or equivalence card of an external program was incorrect.

THE MONITOR DECLARATION

At the programer's option, the compiler may be directed to produce on the LINE PRINTER the results of assignment statements executed at object time. The MONITOR declaration specifies the particular variables to be monitored.

GENERAL FORM:

MONITOR 𝔪£

where 𝔪£ is a monitor list which contains identifiers separated by commas. The monitor list may be enclosed in parentheses if desired. These identifiers specify the variables to be monitored. Specifically, the monitor list may consist of:

Names of simple variables;

Names of arrays (optionally followed by a pair of parentheses);

Names of procedures (optionally followed by a pair of parentheses); and

Statement, subroutine, or segment labels.

Names of simple variables or arrays specify that the compiler is to print the result of any assignment statement which assigns a new value to the listed simple variable or variable with subscripts. Names of procedures or labels of statements, subroutines, or segments specify portions of the compiled program. Any assignment statement within the named portion is monitored, regardless of whether the identifier on the left side of the assignment statement had been specifically named in the monitor list.

Whenever an assignment statement is monitored, the computer will print the first five characters of the identifiers which have been assigned, followed by the value currently computed for those identifiers. In the case of a variable with subscripts, the current values of the subscripts are not shown. The particular element of the array being monitored should be determined by monitoring the subscripts themselves.

EXAMPLE:

MONITOR K, L, A(,), VECTOR

This declaration might cause the printout:

```
K     =    0000000001
L     =    0000000001
A     =    .3794284100,  00
L     =    0000000002
A     =    .1230000000,  04
L     =    0000000003
A     =    .9347281800,–01
VECTO =    0000000064
```

MONITOR declarations must precede all statements of the symbolic program. Depressing PROGRAM CONTROL SWITCH 0 will inhibit the monitoring action at run time.

Symbolic Memory Printout

Whenever a MONITOR declaration appears in a program, the compiler provides for a symbolic storage dump of the portion of memory reserved for the storage of variables, as well as for the monitoring just described.

The LINE PRINTER produces this listing under the headings:

VARIABLE IN PROGRAM VALUE

The label message appears as

LAST LABEL PASSED WAS £

followed by a listing under the headings:

LABEL IN PROGRAM NUMBER OF TIMES EXECUTED

If a program contains a MONITOR declaration, a storage printout can be obtained when the program stops by depressing the RESET-TRANSFER switch. Depressing the START switch after the printout resumes computation.

If the programer desires to use the storage printout feature without monitoring either specific variables or portions of the program, he may write as the first statement of his program the MONITOR declaration, omitting the monitor list.

Statement Monitoring at Object Time

Provision has been made to allow the programer to monitor the control sequence of labeled statements and to obtain a symbolic memory dump at specified points within the program *during the running of the object program*. This is accomplished with the use of statement monitoring control cards which immediately precede the data cards (see APPENDIX A, *Operating Instructions*).

These cards have the form:

COLUMNS	CONTENTS
1	5
2-80	MONITOR £ ﹏;

where £ is the list comprised of the names of labels, or names of labels followed by a parenthesized number, each separated by commas.

EXAMPLE:

MONITOR F, BIG(50), START(1);

If the statement monitoring command precedes the data for a given symbolic program, the first five characters of every label of every labeled statement will be printed out on the LINE PRINTER as the statement is encountered. Also, if the name of some label appears in the list, then a symbolic memory dump will occur on the LINE PRINTER immediately before every execution of that statement or only on the *n*th time the statement is encountered if it is followed by the number *n* in parentheses. Thus, in the previous example, as every labeled statement is encountered, the first five characters of its name are printed on the LINE PRINTER; in addition, every time the label F, the 50th time the label BIG, and the first time the label START is encountered, a symbolic memory dump will be given on the LINE PRINTER.

ERROR MESSAGES FROM LIBRARY PROCEDURES

Errors in library procedures which occur are indicated by error messages on the LINE PRINTER. In the following, ℘ is the name of the procedure in which the error occurred, £ is the first five characters of the label of the last labeled statement passed in the program, and *nnnn* is the number of times this statement has been executed. (However, if the identifier for £ is an integer, it will be printed in its entirety.) An error message will always contain the first portion of the message, e.g.,

RESULT OUT OF RANGE IN ℘

If a MONITOR statement has been given, the message will also contain £(*nnnn*). Note that the error produced may have occurred in an unlabeled statement, executed after the last labeled statement was passed.

RESULT OUT OF RANGE IN ℘ -£ (*nnnn*)

FIXED POINT Result $|R| \geq 10^{10}$
FLOATING POINT $|R| \geq 0.99999999 \times 10^{49}$

RESULT UNDEFINED FOR ℘ -£ (*nnnn*)

An attempt has been made to determine log x for $x \leq 0$; to determine $\sqrt{-x}$, etc.

RESULT ILL-DEFINED FOR ℘ -£ (*nnnn*)

An attempt has been made to determine $\sin x$ for $|x| \geq 10^7$, etc.

ARITHMETIC OVERFLOW ℘ -£ (*nnnn*)

In some statement previous to the execution of £, an attempt has been made to divide by zero; or to determine one of the following:

FIXED POINT $R = |x_1 \pm x_2| > 10^{10}$
FLOATING POINT $R = |x_1 \pm x_2| > 0.99999999 \times 10^{49}$
FLOATING POINT $R = |x_1 \cdot x_2| > 0.99999999 \times 10^{49}$
FLOATING POINT $R = |x_1 / x_2| > 0.99999999 \times 10^{49}$

LISTING OF THE COMPILED PROGRAM

The programer can obtain a machine-language listing of the program being compiled by depressing PROGRAM CONTROL SWITCH 2 at compiling time. The use of this feature reduces compiling speed to approximately one-third of normal. This feature has been included as a diagnostic technique for use only when other methods have been exhausted.

In addition to the usual listing of the symbolic program on the LINE PRINTER, the use of this feature produces output of the form: *T B A L llll s bbbb pp aaaa*

where

T	is the type of line (see below);
B A L	are relocation-control digits for the *bbbb aaaa*, and *llll* fields respectively;
llll	is the location associated with this line; and
s bbbb pp aaaa	is a machine-language word.

The storage allotted for the object program is divided into two portions, either of which can be relocated independently. One of these portions consists of those cells which are allotted to simple and array variables, constants, and temporary storage. The other portion consists of those cells which contain the actual instructions produced by the compiler and by the machine-language procedures.

The control field *bbbb*, the address field *aaaa*, and the location *llll* may each be relocated relative to either portion of storage, or may be specified as being absolute. The relocation digits 0, 1, and 2 specify that a field is to be absolute, relative to the variable region, or relative to the instruction region respectively.

The relocation constant for the instructions is 0000. The relocation constant for the variables is printed out at the end of compilation, as described on page A-1.

If the indicator *T* for the type of line is zero, the entire word will be stored in the cell in relative position *llll*; if *T* is one, only the address portion will be stored. This is the technique used to provide references to labels, the relative locations of which are not yet known when they are referred to in the symbolic program.

ORGANIZATION OF PROGRAMS
HARMONIC-BOUNDARY VALUES
SURVEY TRAVERSE CALCULATIONS
OPTICAL RAY-TRACING
HOUSEHOLDER REDUCTION
CROUT'S METHOD

XI . . .

programs in algol

THE OVER-ALL ORGANIZATION of programs written in this representation of ALGOL is a matter which the rules leave largely to the preference of the individual programer. However, there are rules of precedence which must be observed regarding the interrelations among certain declarations and statements. Below is a summary of such rules, arranged in the form of a suggested program outline.

MONITOR declaration—appears only if one or more variables or statements are to be monitored, or if the programer wishes to provide for the possibility of a memory dump (see page 10-3); it must be the first declaration (other than a COMMENT declaration) of the symbolic program.

Declarations of type—declare the types of any identifiers which are not floating-point (see CHAPTER V); they must precede use of the identifier in a statement.

ARRAY declarations—reserve storage for multidimensional arrays of data; they must precede use of any variable which is an element of an array.

PROCEDURE and FUNCTION declarations (symbolic and external)—make specific subprograms available to the compiler; they must appear prior to the first use of the subprograms.

Operational statements—Assignment statements and control statements.

INPUT and OUTPUT declarations—may appear anywhere in the symbolic program.

FORMAT declarations—may appear anywhere in the symbolic program.

FINISH declaration—must appear at the end of the symbolic program.

Machine-language programs—may be used in conjunction with a symbolic program. The latter must contain definitions of all the machine-language programs so used. These programs are included after the FINISH declaration of the symbolic program, and must themselves be terminated by a second FINISH statement.

COMMENT declarations—affect neither the compilation nor the object program, and may appear anywhere.

EXAMPLES OF PROGRAMS

The following complete programs are presented as illustrations of the rules delineated in this manual.

J. G. Herriot, of Stanford University, has written the following program to determine an approximation of harmonic-boundary values, using orthonormal functions.

```
COMMENT THIS PROGRAM FIRST CONSTRUCTS A SET OF
  ORTHONORMAL FUNCTIONS AND THEN USES THEM TO
  FIND AN APPROXIMATION TO THE SOLUTION OF A
  HARMONIC BOUNDARY-VALUE PROBLEM;
COMMENT WE FIRST CONSTRUCT THE ORTHONORMAL
  FUNCTIONS;
INTEGER I, J, K, L, M, N, NU, TH;
ARRAY R(29), HFN(29), DSUM(24), HFCN(5), HFCEN(6),
 FA(25,25), A(25,25), B(25,25), HA(47), HAA(24);
INPUT DATA (FOR I = (1,1,29); R(I)), DIMEN(N);
OUTPUT FRESULTS (FOR I = (1,1,N); FOR J = (1,1,N); FA(I,J)),
  ARESULTS (FOR I = (1,1,N); FOR J = (1,1,N); A(I,J)),
  BRESULTS (FOR I = (1,1,N); FOR J = (1,1,N); B(I,J)),
  COEFFS (FOR NU = (4,4,N - 1); HA (2NU - 1)),
  HFNRES (FOR K = (1,1,29); HFN(K)),
CRES(CONST), HFCNRES (TH, FOR K = (1,1,5); HFNC(K)),
  HFCENRES (TH, FOR K = (1,1,6); HFCEN(K));
```

```
FORMAT VECTOR (B8,6F16.8,W0),
  FTITLE (B48,*FRESULTS,FA(I,J)*,W3,W2),
  ATITLE (B48,*ARESULTS,A(I,J)*,W3,W2),
  COEFTITLE (B30,*HA(8NU − 1)*,W2),
  BDYVALUES (B42,*PRELIMINARY BOUNDARY VALUES*, W3,
    W2),
  CBDYVALUES (B43,*CORRECTED BOUNDARY VALUES*,W2),
  CONTITLE (B50,*CONSTANT*,W2),
  TABLE (B8,I2,B6,6F16.8,W0),
  TABLEHEAD (B40, *THE VALUES OF H(RHO,TH) IN B*, W3,
    W2),
  TABLELINE (B13,*RHO*,B6,*0.5*,B13,*1.0*,B13,*1.5*,B13,
  *2.0*,B13,*2.5*,B13,*3.0*,W0),
  TABLETH (B8,*TH*,W0);
START..READ (;;DATA); RDIM..READ (;;DIMEN);
  FOR I = (1,1,N); FOR J = (I,4,N);
BEGIN L = I − J; K = I + J;
  SUM = R (1)*K + 1.5.R(18)*K.COS(0.59341195.L)
    + 0.5.R(29)*K.COS(0.78539816.L);
  FOR M = (2,1,17);
  SUM = SUM + 2.0.R(M)*K.COS((M − 1).0.034906585.L);
  FOR M = (19,1,28);
  SUM = SUM + R(M)*K.COS((0.59341195 + (M − 18)
    .0.017453293).L);
  FA(I,J) = (8.0/K).0.017453293.SUM END;
    WRITE (;;FTITLE);
WRITE (;; FRESULTS, VECTOR);
  FOR J = (1,1,N); B(1,J) = FA(1,J);
  FOR I = (2,1,N);
    BEGIN FOR J = (1,1,I − 1);
      B(I,J) = −B(J,I)/B(J,J);
    FOR J = (I,1,N);
      BEGIN B(I,J) = FA(I,J);
    FOR K = (1,1,I − 1);
    B(I,J) = B(I,J) + B(I,K).B(K,J) END;
  FOR J = (1,1,I − 1);
      B(I,J) = B(I,J).SQRT(B(J,J)/B(I,I)) END;
FOR I = (1,1,N); B(I,I) = 1.0/(SQRT(B(I,I)).I);
WRITE (;;BTITLE);
WRITE (;;BRESULTS, VECTOR);
FOR I = (1,1,N); FOR J = (1,1,N); A(I,J) = 0;
  A(1,1) = B(1,1);
FOR I = (2,1,N);
  BEGIN FOR J = (1,1,I − 1);
    BEGIN A(I,J) = 0;
  FOR K = (J,1,I − 1);
  A(I,J) = A(I,J) + B(I,K).A(K,J) END;
    A(I,I) = B(I,I) END; WRITE (;;ATITLE);
WRITE (;;ARESULTS, VECTOR);
COMMENT NOW CONSTRUCT THE APPROXIMATION TO
THE SOLUTION;
  FOR J = (4,4,N − 1);
    BEGIN DSUM(J) = 0;
  FOR M = (1,1,17);
    DSUM(J) = DSUM(J) + (R(M)*2 + R(M + 1)*2).
    (R(M + 1)*J.SIN(M.0.034906585.J)
    −R(M)*J.SIN((M − 1).0.034906585.J));
  FOR M = (18,1,28);
  DSUM(J) = DSUM(J) + (R(M)*2 + R(M + 1)*2.(R(M + 1)
    *J.SIN((0.59341195 + (M − 17).0.017453293).J)
    − R(M)*J.SIN((0.59341195 + (M − 18).0.017453293)
      .J)) END;
  FOR NU = (4,4,N − 1); BEGIN HA(2NU − 1) = 0;
    FOR J = (4,4,NU);
    HA(2NU − 1) = HA(2NU − 1) + A(NU,J).DSUM(J);
    HA(2NU − 1) = 4.0.HA(2NU − 1) END;
      WRITE(;;COEFTITLE);
  WRITE (;;COEFFS, VECTOR);
  FOR J = (4,4,N − 1); BEGIN HAA(J) = 0;
    FOR NU = (J,4,N − 1);
  HAA(J) = HAA(J) + HA(2NU − 1).A(NU,J) END;
FOR M = (1,1,18); BEGIN HFN(M) = 0;
  FOR J = (4,4,N − 1);
  HFN(M) = HFN(M) + HAA(J).R(M)*J.COS((M − 1)
    .0.034906585.J) END;
FOR M = (19,1,29); BEGIN HFN(M) = 0;
FOR J = (4,4,N − 1);
  HFN(M) = HFN(M) + HAA(J).R(M)*J.COS((0.59341195
    + (M − 18).0.017453293).J) END;
WRITE (;;BDYVALUES);
WRITE (;;HFNRES, VECTOR);
AVT = 0;
FOR M = (1,1,29); AVT = AVT + R(M)*2 − HFN(M);
  CONST = AVT/29.0; WRITE (;;CONTITLE);
  WRITE (;;CRES, VECTOR);
FOR M = (1,1,29); HFN(M) = CONST + HFN(M);
WRITE (;;CBDYVALUES);
WRITE (;;HFNRES, VECTOR);
FOR I = (1,1,5); BEGIN TH = 5.(I − 1);
  FOR J = (1,1,5);
BEGIN HFCN(J) = CONST;
FOR M = (4,4,N − 1);
HFCN(J) = HFCN(J) + HAA(M).(0.5.J)*M.
COS((I − 1).0.087266463.M) END;
  WRITE(;;TABLEHEAD);
  WRITE(;;TABLELINE);
  WRITE(;;TABLETH);
WRITE (;;HFCNRES, TABLE) END;
FOR I = (6,1,10); BEGIN TH = 5.(I − 1);
  FOR J = (1,1,6);
BEGIN HFCEN(J) = CONST;
FOR M = (4,4,N − 1);
HFCEN(J) = HFCEN(J) + HAA(M).(0.5.J)*M.COS((I − 1)
    .0.087266463.M) END;
WRITE (;;HFCENRES, TABLE) END;
STOP 1234;
GO TO RDIM;
  FINISH;
```

The program which follows is one for survey traverse calculations.

```
COMMENT SURVEY TRAVERSE CALCULATIONS;
INTEGER I, J, K, SURVEY, D(), M(), S(), Q(), N;
FUNCTION LENGTH(X,Y) = SQRT(X*2 + Y*2);
ARRAY D(200),M(200), S(200), Q(200), MD(200), NS(200),
  EW(200), CNS(201), CEW(201);
  START.. READ (;;IDENT); TMD = 0; TNS = 0; TEW = 0;
FOR I = (1,1,N); BEGIN
READ (;;STATION); IF I NEQ K; STOP K;
Z = (60(60D(I) + M(I)) + S(I))/6.48**5;
SWITCH Q(I), (QUAD1, QUAD2, QUAD3, QUAD4);
QUAD1.. Z = 0.5 - Z; GO TO ANGLE;
QUAD2.. Z = 1.5 + Z; GO TO ANGLE;
QUAD3.. Z = 0.5 + Z; GO TO ANGLE;
QUAD4.. Z = 1.5 - Z;
ANGLE.. ALPHA = 3.1415927Z;
NS(I) = MD(I)SIN(ALPHA); TNS = TNS + NS(I);
EW(I) = MD(I)COS(ALPHA); TEW = TEW + EW(I);
TMD = TMD + MD(I) END;
ERROR = LENGTH (TNS, TEW); WRITE (;;TITLE, F1);
NSC = -TNS/TMD; EWCF = -TEW/TMD; TCD = 0;
  TCNS = 0; TCEW = 0;
FOR I = (1,1,N); BEGIN
CNS(I) = NS(I) + MD(I).NSCF; TCNS = TCNS + CNS(I);
CEW(I) = EW(I) + MD(I).EWCF; TCEW = TCEW + CEW(I);
CD = LENGTH(CNS(I), CEW(I)); TCD = TCD + CD;
WRITE (;;ANSWERS,F2) END;
CNS(N + 1) = CNS(1); CEW(N + 1) = CEW(1); SUM = 0;
FOR I = (1,1,N); SUM = SUM + (CNS(I + 1) - CNS(I))
  (CEW(I + 1) + CEW(I));
SQFT = ABS(SUM)/2; ACRES = SQFT/43560;
WRITE (;;TOTALS, F3); GO TO START;
INPUT IDENT(SURVEY, N), STATION(K,D(I), M(K), S(I),
  Q(I), MD(I));
OUTPUT TITLE(SURVEY, N, ERROR),
ANSWERS (I, D(I), M(I), S(I), Q(I), MD(I), CD, CNS(I), CEW(I)),
TOTALS (TMD, TCD, TCNS, TCEW, SQFT, ACRES);
FORMAT F1(*SURVEY*, I8, B5, *NUMBER OF LEGS*, I5,
  *CLOSURE ERROR*, X9.2, W1, *LEG*, B
5, *ANGLE*, B7, *MEASURED*, B5, *CORRECTED*, B3,
  *NORTH-SOUTH EAST-WEST*, W6, *NO. DD
MM SS Q DISTANCE DISTANCE DISPLACEMENT
  DISPLACEMENT*, 2W), F2,(I3, I5, 2I3, I2, 4X13.2, W),
F3(B6, *TOTALS*, B4, 4X13.2,W4, *AREA OF TRAVERSE*,
  X13.2,*SQUARE FEET*,
X13.2,*ACRES*, W6);
FINISH;
123456  4
1  45 01 30 2  1000.00
2  46 13 24 4  2003.69
3  43 12 02 3   995.28
4  45 25 17 1  2001.64
```

The following program for optical ray-tracing was written by R. Mitchell.

```
COMMENT OPTICAL TRACE PROGRAM, R. F. MITCHELL,
  VIDYA 1;
INTEGER M,J,K,JA,JB;
ARRAY A(4),B(4),C(4);
A(1) = -260.0; A(3) = -600.0; A(4) = 0.0;
ARRAYG(6) = (25.0,50.0,75.0,100.0,125.0,150.0);
C(1) = 1.0; C(2) = 3.436; C(3) = 1.0; C(4) = -1.0;
B(2) = 339.75;  B(3) = 1.0; B(4) = 0.0;
ARRAY B1(6) = (5.5,5.75,6.0,6.25,6.75,7.0);
FOR JB = (1,1,6); BEGIN  B(1) = B1 (JB);
A(2) = A(1) - B1(JB)(1.0 - 1.0/C(2).C(2));
WRITE (; ;PARAM,F6);
OUTPUT PARAM (FOR K = (1,1,4); (A(K),B(K),C(K)));
FORMAT F6(W3,11(3F20.8,W4));
FOR M = (1,1,6); BEGIN H1 = G(M);
WRITE (; ;GVALU,F1);
OUTPUT GVALU(H1);
FORMAT F1 (B10,*G*,X20.8,W6);
SUMD = 0.0; E = 0.0; P = H1;
FOR J = (1,1,3); BEGIN
R1 = P/A(J);
IF ABS(R1) GTR 0.95; GO TO PRINT;
I = ARCSIN(R1);
R2 = R1. C(J)/C(J + 1);
IF ABS(R2) GTR 0.95; GO TO PRINT;
IP = ARCSIN(R2);
E = E + I - IP;
H2 = A(J + 1)SIN(E + I);
R3 = SIN(E);
IF R3 = SIN(E);
IF R3 EQL 0.0; R3 = 1.0**-20;
DL = (H2 - H1)/R3;
DU = DL.C(J + 1);
SUMD = SUMD + DU;
H1 = H2;
WRITE(;;EDH,F2);
OUTPUT EDH(E,DU,H1);
FORMAT F2(B5,*E*,X15.8,B10,*D*,X15.8,B10,*H*,X15.8,W4);
IF (PCS(1)); BEGIN
WRITE (;;ALL,F5);
OUTPUT ALL(R1,R2,I,IP,E,H2,R3,DL,DU,P);
FORMAT F5(2(5F12.5,W4))  END;
P = A(J).R2 - R3(A(J + 1) - A(J) + B(J)) END;
L = A(3)(1.0 + R2/R3);
SUMD = SUMD - DU;
WRITE (;;LSUMD,F3);
OUTPUT LSUMD(L,SUMD);
FORMAT F3(B10,*L*,X20.8,B10,*SUMD*,X20.8,W6,W4) END
  END; STOP;
PRINT..WRITE (;;ERROR,F);
OUTPUT ERROR(R1,R2,A(J),C(J + 1),P);
FORMAT F(*R1*,X9.4,B4,*R2*,X9.4,B4,*A*,X9.4,B4,*C*,X9.6,
  B4,*P*,X9.4,W1);
STOP; FINISH;
```

The short program which follows is for a reduction of a square matrix to tridiagonal form, using the method of Householder.

```
COMMENT HOUSEHOLDER REDUCTION TO TRIDIAGONAL
  FORM;
INTEGER I, J, K, L, R, N; ARRAY A (50,50), X(50), P(50);
N = 5;
IN.. READ(;;ELEMENT); IF I NEQ 0; BEGIN A(I,J) = Q;
  GO TO IN END;
FOR R = (1,1,N - 1); BEGIN WRITE (;;AOUT, AF); L = R + 1;
S = 0; FOR J = (L,1,N); S = S + A(R,J)*2;
  S = SIGN (A(R,L))/2SQRT(S);
WRITE (;;BOUT, BF);
X(L) = SQRT(0.5 + A(R,L).S); S = S/X(L);
FOR J = (R + 2,1,N); X(J) = S.A(R,J);
FOR J = (R,1,N); BEGIN S = 0; FOR K = (L,1,N);
S = S + A(MIN(J,K), MAX(J,K)).X(K); P(J) = S END;
S = 0; FOR J = (L,1,N); S = S + K(J).P(J);
FOR J = (L,1,N); P(J) = P(J) - S.X(J);
FOR J = (L,1,N); FOR K = (J,1,N); A(J,K) = A(J,K) - 2(X(J).
  P(K) + X(K).P(J)) END;
WRITE (;;AOUT, AF); STOP; GO TO IN;
INPUT ELEMENT (I,J,Q); OUTPUT AOUT (A(R,R)),
  BOUT (-0.5/S);
FORMAT AF(B10, X10.5, W), BF(B40, X10.5,W);
FINISH;
```

The program below has been written by G. Forsythe, of Stanford University. It solves a set of linear equations of the form **Ay** = **B**, using Crout's method with interchanges.

```
COMMENT FORSYTHE PROGRAM;
PROCEDURE PRODUCT (;N, A( ), P, E);
BEGIN
COMMENT THIS FORMS THE PRODUCT OF ARBITRARY FLOAT-
    ING NUMBERS A(I),
  FOR I = (1,1,N). EXPONENT OVERFLOW OR UNDERFLOW IS
  PREVENTED. THE ANSWER IS P TIMES 10*E, WHERE E IS 0
  IF POSSIBLE. IF E NEQ 0, THEN WE NORMALIZE P SO THAT
  0.1 LEQ ABS(P) LSS 1.0;
INTEGER E, F, I, K, N;
Q = 1.0**-10; F = 10;
FOR I = (1,1,N);
BEGIN IF A(I) EQL 0.0;
  BEGIN P = 0.0; E = 0; RETURN END;
  IF ABS(A(I) ) LEQ 1.0;
  BEGIN F = F - 20; Q = Q.(10.0*20) END;
  Q = Q.A(I); X = ABS(Q);
  FOR K = (-10,1,10), (-11, -1, -41), (11,1,41);
  IF ( (10.0*K LEQ X) AND (X LSS 10.0*(K + 1) ) );
BEGIN Q = Q.(10.0*(-10 - K) ); F = F + K + 10; GO TO 1
    END;
1.. END;
  IF ( ( (-40) LEQ F) AND (F LEQ 58) );
  BEGIN P = (Q.(10.0*9) ).(10.0*(F - 9) ); E = 0; RETURN
    END;
  P = Q.(10.0*9); E = F - 9; RETURN END PRODUCT ( );
PROCEDURE INNERPRODUCT (S,F,U( ), V( ) );
BEGIN COMMENT THIS FORMS THE INNER PRODUCT OF THE
    VECTORS U(I) AND V(I) FOR I = (S,1,F);
  INTEGER I, S, F; SUM = 0.0;
  FOR I = (S,1,F); SUM = SUM + U(I).V(I);
  INNERPRODUCT( ) = SUM; RETURN END INNERPRODUCT();
PROCEDURE CROUT4(;N, A(,), B( ), Y( ), PIVOT( ), DET, EX7;
SINGULAR, IP( ) );
BEGIN COMMENT THIS IS CROUTS METHOD WITH INTE-
CHANGES, TO SOLVE AY = B AND OBTAIN THE TRIANGULAR
DECOMPOSITION. IP( ) STANDS FOR AN INNER PRODUCT
ROUTINE THAT MUST BE AVAILABLE WHEN CROUT4( ) IS
CALLED. ALSO, PRODUCT ( ) MUST BE AVAILABLE. THE
DETERMINANT OF A IS COMPUTED IN THE FORM DET TIMES
10*EX7, WHERE EX7 IS 0 IF POSSIBLE. IF EX7 NEQ 0, THEN
WE NORMALIZE DET WITH 0.1 LEQ ABS(DET) LSS 1;
INTEGER K, I, J, IMAX, N, PIVOT; INTEGER EX7; INT = 1.0;
FOR K = (1,1,N);
BEGIN TEMP = 0; FOR I = (K,1,N);
  BEGIN A(I,K) = A(I,K) - IP(1, K - 1, A(I, ), A( ,K) );
  IF ABS(A(I,K) ) GTR TEMP;
  BEGIN TEMP = ABS(A(I,K) ); IMAX = I END END;
  PIVOT(K) = IMAX;
COMMENT WE HAVE FOUND THAT A(IMAX,K) IS THE LARGEST
PIVOT IN COL K. NOW WE INTERCHANGE ROWS K AND IMAX;
IF IMAX NEQ K; BEGIN INT = -INT; FOR J = (1,1,N);
  BEGIN TEMP = A(K,J); A(K,J) = A(IMAX,J);
  A(IMAX,J) = TEMP END;
  TEMP = B(K); B(K) = B(IMAX); B(IMAX) = TEMP END;
COMMENT NOW FOR THE ELIMINATION;
IF A(K,K) EQL 0.0;
BEGIN DET = 0.0; EX7 = 0; GO TO SINGULAR END;
  FOR I = (K + 1,1,N);
  BEGIN XX = A(I,K); XY = A(K,K); X = 1.0; X = X.X;
    A(I,K) = XX/XY END;
  FOR J = (K + 1,1,N); A(K,J) = A(K,J) - IP(1, K - 1, A(K,),
    A(,J) );
  B(K) = B(K) - IP(1, K - 1, A(K,), B( ) ) END;
  FOR I = (1,1,N); Y(I) = A(I,I);
  PRODUCT (; N, Y( ), DET, EX7); DET = INT. DET;
COMMENT NOW FOR THE BACK SUBSTITUTION;
FOR K = (N,-1,1);
BEGIN XX = B(K) - IP(K + 1, N, A(K,), Y( ) ); XY = A(K,K);
  X = 1.0; X = X.X; Y(K) = XX/XY END; RETURN END CROUT4
    ( );
PROCEDURE SOLV2(; N, B(,), C( ), PIVOT( ), Z( ); IP( ) );
BEGIN
COMMENT IT IS ASSUMED THAT A MATRIX A HAS ALREADY
BEEN TRANSFORMED INTO B BY CROUT, BUT THAT A NEW
COLUMN C HAS NOT BEEN PROCESSED. SOLV2( ) SOLVES
THE SYSTEM BZ = C. AN INNERPRODUCT PROCEDURE MUST
BE USED WITH SOLV2( );
```

```
  INTEGER K, N, PIVOT;
 FOR K = (1,1,N);
 BEGIN TEMP = C(PIVOT(K) ); C(PIVOT(K) ) = C(K);
  C(K) = TEMP; C(K) = C(K) − IP(1, K − 1, B(K,), C( ) );
    END;
  FOR K = (N, −1, 1); Z(K) = (C(K) − IP(K + 1, N, B(K,),
    Z( ) ) )/B(K,K);
  RETURN END SOLV2( );
COMMENT FORSYTHE TEST CROUT US169 EXT 2274;
FORMAT FRMTFL(W0, (6F19.8, W0) );
FORMAT FRMTFX(W0, (6I19, W0) );
INTEGER PIVOT ( ); INTEGER EX; INTEGER I, J, N;
ARRAY A(70,70), B(70), Y(70), C(70), PIVOT(70);
INPUT DATA(N, FOR I = (1,1,N); (FOR J = (1,1,N); A(I,J),
  B(I) ) );
INPUT VECTOR(N, FOR I = (1,1,N); C(I) );
START.. READ(; ; DATA); READ(; ; VECTOR); OUTPUT ORDER
  (N);
OUTPUT DATAO (FOR I = (1,1,N); (FOR J = (1,1,N); A(I,J),
  B(I) ) );
OUTPUT VECTORO (FOR I = (1,1,N); C(I) );
WRITE (; ; ORDER, FRMTFX);
WRITE (; ; DATAO, FRMTFL);
WRITE (; ; ORDER, FRMTFX);
WRITE (; ; VECTORO, FRMTFL);
CROUT4 (; N, A(,), B( ), Y( ), PIVOT( ), DET, EX;
  SINGULAR, INNERPRODUCT( ) );
WRITE (; ; DATAO, FRMTFL);
OUTPUT ANSWER (FOR I = (1,1,N); Y(I) );
OUTPUT PIVOTO (N, FOR I = (1,1,N); PIVOT (I) );
OUTPUT DETO (DET);
OUTPUT EXPO (EX);
WRITE (; ; PIVOTO, FRMTFX);
WRITE (; ; ANSWER, FRMTFL);
WRITE (; ; DETO, FRMTFL);
WRITE (; ; EXPO, FRMTFX);
SOLV2 (; N, A(,), C( ), PIVOT( ), Y( ); INNERPRODUCT( ) );
WRITE (; ; VECTORO, FRMTFL);
WRITE (; ; ANSWER, FRMTFL);
GO TO START; SINGULAR..WRITE (; ; FRMTSI);
FORMAT FRMTSI (W0, *SINGULAR*, W0); GO TO START;
    FINISH;
```

APPENDIX A

operating instructions

THIS APPENDIX CONTAINS the information required for the maintenance and operation of the compiler. It is assumed that the user is in possession of the magnetic-tape reel containing the compiler, and of the card decks containing the following routines: COMPILER CALLOUT, COMPILED PROGRAM CALLOUT, DUMP CALLOUT, COMPILER DUPLICATION CALLOUT, and LIBRARY PROCESSOR CALLOUT.

The compiler system is contained on a reel of magnetic tape which is to be mounted on TAPE STORAGE UNIT 2 whenever it is used. This tape contains the compiler routine proper, the libarary routines, and a collection of routines which should be useful in maintaining the compiler and compiled programs.

For the most part, the compiler system is controlled by means of decks of cards which load the desired routines from the compiler tape and then transfer control to these routines. These decks are termed 'callout decks.' Any callout deck is read by placing it in the CARD READER (INPUT UNIT 1) and executing a CARD READ command: 0 1000 60 *xxxx*. (The address *xxxx* is irrelevant.) These decks need no blank cards preceding or following them except as specifically noted below.

Preparation of Symbolic Decks

Decks of symbolic compiler language are punched with the digit two in column 1 of each card. The statements to be compiled occupy columns 2 through 72.

The symbolic deck is constructed by assembling the following card deck:

First: The COMPILER CALLOUT deck;

Second: The ALGOL statements;

Third: External machine-language programs (if any);

Fourth: STATEMENT MONITORING CONTROL cards (if any; see CHAPTER X, *Diagnostic Facilities*);

Fifth: Input data cards (if any), and

Sixth: Two blank cards, or any number of 'reject' cards (i.e., cards with the digit seven punched in column 1).

Compiling a Program

Mount a scratch tape which has been preblocked into 100-word blocks on both lanes, designate the tape unit as '1,' and place it in WRITE status. Place the symbolic deck (as described above) in the CARD READER and execute a CRD (0 1000 60 *xxxx*). The compiler will read cards and produce a copy of them on the LINE PRINTER (OUTPUT UNIT 2) along with any error messages resulting from the compilation. Meanwhile, the compiled program is written on the tape on TAPE STORAGE UNIT 1. After compilation is complete, the following two messages are produced on the LINE PRINTER:

COMPILED PROGRAM ENDS AT *mmmm*
NEXT AVAILABLE CELL IS *nnnn*

where *mmmm* and *nnnn* are absolute addresses. The variables for the program are stored in memory between the end of the compiled program (*mmmm*) and the next available cell (*nnnn*).

In addition to the above cell-count message, the A register will display either:

O K (0757 00 7250)
provided that no error messages were produced, or
X X (0525 00 5250)
in the event that the compiler detected errors.

If the compilation has been properly completed, depressing the START switch will load and execute the program.

Operation of the FINISH Statement

When the compiler encounters the FINISH declaration, it writes a HALT instruction followed by a CARD READ instruction at the end of the object-language program, adds the library procedures and then stops compilation. Depressing the START key then initiates execution of the object program. Upon completion of this object-language program, pressing the START key executes the CARD READ instruction, as the first step in the compilation of another card deck.

If the programer wishes to avoid this sequence of events, he should precede the FINISH declaration in his symbolic program with a STOP statement, followed by a statement which transfers control back to the desired point in his program. (See CHAPTER VI.)

Deferred Execution of a Compiler Program

If desired, the magnetic tape containing the compiled program may be remounted on TAPE STORAGE UNIT 1 at some later time and the program loaded and run. This is accomplished by placing the deck composed of:

First: The COMPILED PROGRAM CALLOUT cards;

Second: STATEMENT MONITORING CONTROL cards (if any; see CHAPTER X);

Third: Input data (if any); and

Fourth: Two blank cards, or any number of 'reject' cards (i.e., cards with the digit seven punched in column 1).

Dumping a Compiled Program

After a program has been checked out, the compiled program may be converted to cards by placing the deck marked DUMP CALLOUT in the CARD READER and executing a CRD instruction. The program is punched (on OUTPUT UNIT 1) with a suitable loader. The resulting deck, followed by any data cards, if required, and two blank cards, may be placed in the CARD READER and the program run by executing a CRD instruction.

This facility is not available for compiled programs which have used the MONITOR or SEGMENT-OVERLAY features of the compiler.

Duplicating the Compiler-System Tape

It is occasionally desirable to duplicate the compiler tape. To do so, mount on TAPE STORAGE UNIT 1 a tape which has been preblocked with at least 200 blocks of 100 words each on the even lane, and place it in WRITE status. Place the deck marked COMPILER DUPLICATION CALLOUT in the CARD READER and execute a CRD instruction. The compiler program, the service routines, and the entire library will be copied from TAPE STORAGE UNIT 2 to TAPE STORAGE UNIT 1, and the compiler will stop with 7777 00 7777 in the C register. Depressing the START button will cause a card to be read.

Library Maintenance

This section describes the method for placing library procedures on the compiler-system tape. The individual decks which make up the library procedures are described in detail in APPENDIX E, *Construction of Machine-Language Programs.*

Mount the compiler-system tape reel on TAPE STORAGE UNIT 1, and place the unit in NOT WRITE status. On TAPE STORAGE UNIT 2, mount a preblocked tape with at least 200 blocks of 100 words each on the even lane. Put the latter unit in WRITE status. Place the deck marked LIBRARY PROCESSOR CALLOUT in the CARD READER, followed by the desired library procedure decks, a card with the digit two in column 1, and the word FINISH punched on it, followed by two blank cards. Execute a CRD instruction.

If the library processor detects an error, it will produce one of the following error messages on the SUPERVISORY PRINTER.

INCORRECT PUNCTUATION

The ',' on a name card, or the '=' on an equivalence card, was replaced by some other special character, or was missing.

EQUIVALENCE NUMBER TOO LARGE

More than two digits have been given.

MISPLACED NAME CARD

A second name card has appeared prior to the pseudo-operation for FINISH in the machine-language deck of the first.

SEQUENCE ERROR

The addresses of instructions in the machine-language deck being processed are not in the proper order.

MISSING EQUIVALENT

A machine-language instruction with a sign digit of five or six did not have an equivalence card to define its digits $sL = 82$.

The computer will stop with rC displaying 7310 00 0137 after any of the above errors. The processing must then be started over from the beginning.

The message

LIBRARY PROCESSING COMPLETE

is typed out on the SUPERVISORY PRINTER and the computer halts upon completion of updating the library. The new compiler tape on TAPE STORAGE UNIT 2 will then have the processed decks in its library. Depressing the START key will execute a CRD instruction. The library processor routine replaces the entire library with a new one each time it is run. This is no inconvenience as even a large library requires only relatively small numbers of cards.

APPENDIX B

list of reserved identifiers

The list below includes all those identifiers to which the compiler attaches a fixed meaning. These identifiers have been mentioned separately throughout this manual, but are listed here for quick reference. A reserved identifier may not be used by the programer for any purpose other than its function as employed by the compiler. *In addition to this list, the names of all the functions in the library should be considered as reserved identifiers.*

ABS	BOOLEAN	ENTER	FINISH	LEQ	PCS
AND	COMMENT	EQIV	FLOATING	LSS	PROCEDURE
ARRAY	EITHER	EQL	FOR	MAX	REAL
BEGIN	END	EXTERNAL	FORMAT	MIN	RETURN
			FUNCTION	MOD	SEGMENT
			GEQ	MONITOR	SIGN
			GO	NEQ	STATEMENT
			GTR	NOT	STOP
			IF	OR	SUBROUTINE
			IMPL	OTHERWISE	SWITCH
			INPUT	OUTPUT	TO
			INTEGER	OVERLAY	UNTIL

APPENDIX C

syntactical description of the compiler language

FORM OF DEFINITIONS

This appendix lists the syntactical definitions which are provided for reference purposes. While it is not the intent to express here every rule possible for the construction of symbolic programs, these definitions should serve to answer many questions which arise concerning the language.

The definitions given here are expressed in a notation which is particularly well suited for syntactical description. A definition has the general form:

Thing being defined ~ *definition*

(the symbol ~ being read as *has the form of*).

The symbols ⟨ and ⟩ are used as brackets which enclose a construction defined elsewhere. The symbol | is to be read as *or*. Other symbols represent themselves.

For example, the definition:

scale factor ~ **⟨integer constant⟩ |
**+⟨integer constant⟩ | **−⟨integer constant⟩

is to be interpreted as meaning: *A scale factor consists of two asterisks perhaps followed by a + or − sign and then followed by an integer constant.*

Syntactical definitions are recursive, that is, the definition may be applied over and over again. For example,

integer constant ~ ⟨digit⟩ |
⟨integer constant⟩ ⟨integer constant⟩

would indicate that an integer constant consists of a string of digits.

It is impossible in these definitions to give the restrictions on the definition. For the latter, it will be necessary to refer to the relevant portion of the text; e.g., an integer is restricted to a maximum of ten digits (in this case, a result of the word length of the BURROUGHS 220).

letter ~ A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

digit ~ 0|1|2|3|4|5|6|7|8|9

identifier ~ ⟨letter⟩ | ⟨identifier⟩ ⟨letter⟩ | ⟨identifier⟩ ⟨digit⟩

integer constant ~ ⟨digit⟩ | ⟨integer constant⟩ ⟨integer constant⟩

basic floating-point constant ~ ⟨integer constant⟩.⟨integer constant⟩

scale factor ~ **⟨integer constant⟩ | **+⟨integer constant⟩ | **−⟨integer constant⟩

floating-point constant ~

⟨basic floating-point constant⟩ |

⟨basic floating-point constant⟩ ⟨scale factor⟩ |

⟨integer⟩ ⟨scale factor⟩

arithmetic constant ~ ⟨integer constant⟩ | ⟨floating-point constant⟩

Boolean constant ~ 0 | 1

constant ~ ⟨arithmetic constant⟩ | ⟨Boolean constant⟩

signed constant ~

⟨arithmetic constant⟩ | +⟨arithmetic constant⟩ | −⟨arithmetic constant⟩

simple variable ~ ⟨identifier⟩

subscript list ~ ⟨arithmetic expression⟩ | ⟨subscript list⟩, ⟨subscript list⟩

variable with subscripts ~ ⟨array identifier⟩ (⟨subscript list⟩)

variable ~ ⟨simple variable⟩ | ⟨variable with subscripts⟩

arithmetic variable ~ ⟨variable⟩

Boolean variable ~ ⟨variable⟩

label ~ ⟨identifier⟩ | ⟨integer⟩

simple argument list ~ ⟨expression⟩ | ⟨simple argument list⟩, ⟨simple argument list⟩

argument-subscript list ~

⟨empty⟩ |
⟨arithmetic expression⟩ |
⟨argument-subscript list⟩, ⟨argument-subscript list⟩

argument array ~ ⟨array identifier⟩ (⟨argument-subscript list⟩)

input-argument list ~

⟨expression⟩ |
⟨argument array⟩ |
⟨input argument list⟩, ⟨input argument list⟩

input-argument portion ~ ⟨empty⟩ | ⟨input-argument list⟩

output-argument list ~

⟨variable⟩ |
⟨argument array⟩ |
⟨output-argument list⟩, ⟨output-argument list⟩

output-argument portion ~ ⟨empty⟩ | ⟨output-argument list⟩

program reference-argument list ~

⟨label⟩ |
⟨procedure identifier⟩ ( ) |
⟨function identifier⟩ ( ) |
⟨program reference-argument list⟩, ⟨program reference-argument list⟩

program reference-argument portion ~ ⟨program reference-argument list⟩

argument list ~

⟨input-argument portion⟩ |
⟨input-argument portion⟩; ⟨output-argument portion⟩ |
⟨input-argument portion⟩; ⟨output-argument portion⟩; ⟨program reference-argument portion⟩

evaluated function ~
⟨function identifier⟩ ((⟨simple argument list⟩)) |
⟨procedure identifier⟩ ((⟨argument list⟩))

arithmetic-evaluated function ~ ⟨evaluated function⟩

Boolean evaluated function ~ ⟨evaluated function⟩

arithmetic operation ~ + | − | · | * | / | ·+ | ·− | *+ | *− | /+ | /−

basic arithmetic expression ~
⟨arithmetic constant⟩ |
⟨arithmetic variable⟩ |
⟨arithmetic evaluated function⟩ |
⟨basic arithmetic expression⟩ ⟨arithmetic operation⟩ ⟨basic arithmetic expression⟩ |
((⟨arithmetic expression⟩))

arithmetic expression ~
⟨basic arithmetic expression⟩ |
+⟨basic arithmetic expression⟩ |
−⟨basic arithmetic expression⟩

relational operation ~ GTR | GEQ | EQL | NEQ | LEQ | LSS

arithmetic relation ~ ⟨arithmetic expression⟩ ⟨relational operation⟩ ⟨arithmetic expression⟩

unary Boolean operation ~ NOT

binary Boolean operation ~ AND | OR | EQIV | IMPL

Boolean expression ~
⟨Boolean constant⟩ |
⟨Boolean variable⟩ |
⟨Boolean evaluated function⟩ |
((⟨arithmetic relation⟩)) |
⟨unary Boolean operation⟩ ⟨Boolean expression⟩ |
⟨Boolean expression⟩ ⟨binary Boolean operation⟩ ⟨Boolean expression⟩ |
((⟨Boolean expression⟩))

expression ~ ⟨arithmetic expression⟩ | ⟨Boolean expression⟩

condition ~ ⟨arithmetic relation⟩ | ⟨Boolean expression⟩

program ~ ⟨statement list⟩; FINISH;

statement ~ ⟨operational statement⟩ | ⟨declaration⟩

statement list ~
⟨operational statement⟩ |
⟨statement⟩; ⟨statement list⟩ |
⟨statement list⟩; ⟨statement⟩

operational statement ~

⟨assignment statement⟩ |
⟨go statement⟩ |
⟨enter statement⟩ |
⟨return statement⟩ |
⟨stop statement⟩ |
⟨switch statement⟩ |
⟨overlay statement⟩ |
⟨clause⟩; ⟨operational statement⟩ |
⟨alternative statement⟩ |
⟨compound statement⟩ |
⟨procedure-call statement⟩ |
⟨labeled dummy statement⟩ |
⟨labeled operational statement⟩

arithmetic assignment statement ~ ⟨arithmetic variable⟩ = ⟨arithmetic expression⟩ |

⟨arithmetic variable⟩ = ⟨arithmetic assignment statement⟩

Boolean assignment statement ~ ⟨Boolean variable⟩ = ⟨Boolean expression⟩ |

⟨Boolean variable⟩ = ⟨Boolean assignment statement⟩

procedure-assignment statement ~ ⟨procedure identifier⟩ ( ) = ⟨expression⟩

assignment statement ~

⟨arithmetic assignment statement⟩ |
⟨Boolean assignment statement⟩ |
⟨procedure-assignment statement⟩

go statement ~ GO ⟨statement label⟩ | GO TO ⟨statement label⟩

enter statement ~ ENTER ⟨subroutine label⟩

return statement ~ RETURN

stop statement ~ STOP | STOP ⟨expression⟩

switch list ~ ⟨statement label⟩ | ⟨switch list⟩, ⟨switch list⟩

switch statement ~ SWITCH ⟨expression⟩, (⟨switch list⟩)

overlay statement ~ OVERLAY ⟨segment label⟩

clause ~ ⟨if clause⟩ | ⟨until clause⟩ | ⟨for clause⟩

if clause ~ IF ⟨condition⟩

until clause ~ UNTIL ⟨condition⟩

for clause ~ FOR ⟨arithmetic variable⟩ = ⟨iteration list⟩

iteration list ~

⟨arithmetic expression⟩ |
(⟨arithmetic expression⟩, ⟨arithmetic expression⟩, ⟨arithmetic expression⟩) |
⟨iteration list⟩, ⟨iteration list⟩

ending ~ END |) | END ⟨label⟩ | END ⟨procedure identifier⟩ ( )

alternative statement head ~

EITHER ⟨if clause⟩; ⟨operational statement⟩ |
⟨alternative statement head⟩; OR ⟨if clause⟩; ⟨operational statement⟩

alternative statement tail ~ ⟨ending⟩ |; OTHERWISE; ⟨operational statement⟩

alternative statement ~ ⟨alternative statement head⟩ ⟨alternative statement tail⟩

procedure-call statement ~ ⟨procedure identifier⟩ (⟨argument list⟩)

compound statement ~

BEGIN ⟨statement list⟩ ⟨ending⟩ |
(⟨statement list⟩ ⟨ending⟩

statement label ~ ⟨label⟩

labeled dummy statement ~ ⟨statement label⟩..

labeled operational statement ~ ⟨statement label⟩..⟨operational statement⟩

declaration ~

⟨type declaration⟩ |
⟨array declaration⟩ |
⟨comment declaration⟩ |
⟨subroutine declaration⟩ |
⟨function declaration⟩ |
⟨procedure declaration⟩ |
⟨segment declaration⟩ |
⟨input declaration⟩ |
⟨output declaration⟩ |
⟨format declaration⟩ |
⟨monitor declaration⟩ |
⟨external declaration⟩

type name ~ FLOATING | INTEGER | BOOLEAN | REAL

type-list element ~

⟨simple variable⟩ |
⟨array identifier⟩ |
⟨array identifier⟩ ( ) |
⟨function identifier⟩ |
⟨function identifier⟩ ( ) |
⟨identifier⟩...

type list ~ ⟨type-list element⟩ | ⟨type list⟩, ⟨type list⟩

type declaration ~

⟨type name⟩ ⟨type list⟩ |
⟨type name⟩ OTHERWISE |
⟨type name⟩ (⟨type list⟩)

integer list ~

⟨integer constant⟩ |
⟨integer list⟩, ⟨integer list⟩

array identifier ~ ⟨identifier⟩

basic array-list element ~ ⟨array identifier⟩ ((⟨integer list⟩))

constant list ~

⟨signed constant⟩ |
⟨constant list⟩, ⟨constant list⟩

array list ~

⟨basic array-list element⟩ |
⟨basic array-list element⟩ = ((⟨constant list⟩)) |
⟨array list⟩, ⟨array list⟩

array declaration ~

ARRAY ⟨array list⟩ |
ARRAY ((⟨array list⟩))

comment declaration ~ COMMENT ⟨any set of characters not containing a semicolon⟩

subroutine label ~ ⟨label⟩

subroutine declaration ~

SUBROUTINE ⟨subroutine label⟩; ⟨compound statement⟩

function identifier ~ ⟨identifier⟩

simple parameter list ~

⟨identifier⟩ |
⟨simple parameter list⟩, ⟨simple parameter list⟩

function declaration ~

FUNCTION ⟨function identifier⟩ ((⟨simple parameter list⟩)) = ⟨expression⟩

parameter-subscript list ~

⟨empty⟩ |
⟨parameter-subscript list⟩, ⟨parameter-subscript list⟩

parameter array ~ ⟨identifier⟩ (parameter-subscript list)

procedure identifier ~ ⟨identifier⟩

input-parameter list ~
⟨identifier⟩ |
⟨parameter array⟩ |
⟨input-parameter list⟩, ⟨input-parameter list⟩

input-parameter portion ~ ⟨empty⟩ | ⟨input-parameter list⟩

output-parameter list ~

⟨identifier⟩ |
⟨parameter array⟩ |
⟨output-parameter list⟩, ⟨output-parameter list⟩

output-parameter portion ~ ⟨empty⟩ | ⟨output-parameter list⟩

program reference-parameter list ~

⟨identifier⟩ |
⟨identifier⟩ ( ) |
⟨program reference-parameter list⟩, ⟨program reference-parameter list⟩

program reference-parameter portion ~ ⟨program reference-parameter list⟩

parameter list ~

⟨input-parameter portion⟩ |
⟨input-parameter portion⟩; ⟨output-parameter portion⟩ |
⟨input-parameter portion⟩; ⟨output-parameter portion⟩; ⟨program reference-parameter portion⟩

procedure declaration ~

PROCEDURE ⟨procedure identifier⟩ (⟨parameter list⟩); ⟨compound statement⟩

segment label ~ ⟨label⟩

segment declaration ~

SEGMENT ⟨segment label⟩; ⟨compound statement⟩

input label ~ ⟨label⟩

input-data list ~

⟨variable⟩ |
⟨for clause⟩; ⟨input-data list⟩ |
⟨for clause⟩; (⟨input-data list⟩) |
⟨input-data list⟩, ⟨input-data list⟩

input list ~

⟨input label⟩ (⟨input-data list⟩) |
⟨input list⟩, ⟨input list⟩

input declaration ~

INPUT ⟨input list⟩ |
INPUT (⟨input list⟩)

output label ~ ⟨label⟩

output-data list ~

⟨expression⟩ |
⟨for clause⟩; ⟨output-data list⟩ |
⟨for clause⟩; (⟨output-data list⟩) |
⟨output-data list⟩, ⟨output-data list⟩

output list ~

⟨output label⟩ (⟨output-data list⟩) |
⟨output list⟩, ⟨output list⟩

output declaration ~

OUTPUT ⟨output list⟩ |
OUTPUT (⟨output list⟩)

basic format phrase ~

⟨letter⟩ |
⟨letter⟩ ⟨integer constant⟩ |
⟨letter⟩ ⟨integer constant⟩.⟨integer constant⟩

format phrase ~

*⟨any string of characters not containing an *⟩* |
⟨basic format phrase⟩ |
⟨integer constant⟩ ⟨basic format phrase⟩

format-phrase list ~

⟨format phrase⟩ |
⟨format-phrase list⟩, ⟨format-phrase list⟩ |
(⟨format-phrase list⟩) |
⟨integer constant⟩ (⟨format-phrase list⟩)

format label ~ ⟨label⟩ | ⟨integer⟩

format list ~

⟨format label⟩ (⟨format list⟩) |
⟨format list⟩, ⟨format list⟩

format declaration ~

FORMAT ⟨format list⟩ |
FORMAT (⟨format list⟩)

monitor declaration ~

MONITOR |
MONITOR ⟨type list⟩ |
MONITOR (⟨type list⟩)

⟨external declaration⟩ ~

EXTERNAL PROCEDURE ⟨procedure identifier⟩ (⟨parameter list⟩) |
EXTERNAL PROCEDURE ⟨procedure identifier⟩ (⟨parameter list⟩); ⟨type name⟩ ⟨procedure identifier⟩ |
EXTERNAL STATEMENT ⟨label⟩

APPENDIX D

transliteration rules

THIS APPENDIX presents a summary of equivalences between the elements of the BURROUGHS Algebraic Compiler language, and the elements of ALGOL, the international algebraic language. It is this latter language which is used for the publication of programs written in the former. Transliteration of programs for the BURROUGHS Algebraic Compiler into ALGOL requires a thorough familiarity with ALGOL, the details of which are available in the literature.†

† See *Communications of the ACM*, vol. 1, no. 12, pp. 8-22; and vol. 3, no. 5, pp. 299-313.

	Reference Language	Burroughs Language
1. BASIC SYMBOLS		
a. *Non-delimiters*		
(1) Letters	A ⁓ Z	A ⁓ Z
	a ⁓ z	A ⁓ Z
(2) Digits	0 ⁓ 9	0 ⁓ 9
b. *Delimiters*		
(1) Operators		
Arithmetic	+	+
	−	−
	×	. The multiplication sign may be omitted in certain instances. It is represented on card equipment as a decimal point (.).
	/	/
Relational	<	LSS
	≤	LEQ
	=	EQL
	≥	GEQ
	>	GTR
	≠	NEQ

1. BASIC SYMBOLS (continued)

b. *Delimiters* (continued)

(1) Operators (continued)

Logical	¬	NOT
	∨	OR
	∧	AND
	≡	EQIV
	(⊃)	IMPL
	Logical implication — no equivalent in the reference language.	
Sequential	*go to*	GO TO
	(no equivalent)	SWITCH
	do	(no equivalent)
	return	RETURN
	stop	STOP
	(no equivalent)	FINISH
	for	FOR
	if	IF
	or	OR
	if either	EITHER IF
	or if	OR IF
	(no equivalent)	UNTIL
(2) Separators	(not required)	⟨space⟩
		Spaces must be used to separate contiguous identifiers or an identifier followed by a constant. *Spaces may not be embedded within an identifier or a constant.*
	.	.
	,	,
	:	..
	;	;
		On standard keypunch equipment, the semicolon is represented by $ (dollar sign.)
	:=	=
	=:	(See PROCEDURES)
	→	(no equivalent)
		Related to the DO statement in the reference language.
	10	**
		Power of 10.
(3) Brackets	*begin*	BEGIN
	end	END
	(	(
	)	)
	[	(
	]	)

1. BASIC SYMBOLS (continued)

b. *Delimiters* (continued)

(3) Brackets (continued)

↑	*(
↓	)
	(Parentheses may be omitted in certain instances.)

(4) Declarators

type	INTEGER, BOOLEAN, FLOATING, REAL
array	ARRAY
function	FUNCTION
comment	COMMENT
procedure	PROCEDURE
(no equivalent)	SUBROUTINE
(no equivalent)	INPUT
(no equivalent)	OUTPUT
(no equivalent)	FORMAT
switch	(no equivalent) See SWITCH statement.

2. EXPRESSIONS

Most expressions are self-evident, except those noted below:

a. Numbers	$G.G_{10}\pm G$	$\mathfrak{G}.\mathfrak{G}**\pm\mathfrak{G}$
	G.	$\mathfrak{G}.\mathfrak{0}$
	.G	$0.\mathfrak{G}$
	10^G	$1**\mathfrak{G}$
b. Simple Variables	I	$\mathfrak{g}$
c. Variables with Subscripts	I[C]	$\mathfrak{g}(\mathfrak{C})$
d. Functions	I(R)	$\mathfrak{g}(\mathfrak{R})$
e. Arithmetic Expressions	(*See* Arithmetic Operators, *above*).	
	$E_1 \uparrow E_2 \downarrow$	E1*(E2) (Parentheses may be omitted in certain instances.)
f. Boolean Expressions	(*See* Relational Operators, *above*).	

3. STATEMENTS

a. Assignment	V := E	$\mathfrak{V} = \mathfrak{E}$
b. Go to	*go to* D	GO TO D
c. Switch	(no equivalent)	SWITCH $\mathfrak{E}$, ($\mathfrak{N}_1$, $\mathfrak{N}_2$, ⋯, $\mathfrak{N}_n$)
d. If	*if* B	IF $\mathfrak{B}$
e. Until	(no equivalent)	UNTIL $\mathfrak{C}$

3. STATEMENTS (continued)

f. For	*for* V := C *for* V := E_{i_1} (E_{s_1}) E_{e_1} ⁓	FOR V = $\mathfrak{e}$ FOR V = ($\mathcal{E}_{i_1}$, $\mathcal{E}_{s_1}$, $\mathcal{E}_{e_1}$)
g. Alternative	*if either* B_1, Σ_1; *or if* B_2, Σ_2; ⁓ *end*	EITHER IF $\mathfrak{B}_1$; $\mathcal{S}_1$; OR IF $\mathfrak{B}_2$; $\mathcal{S}_2$ END EITHER IF $\mathfrak{B}_1$; $\mathcal{S}_1$; OR IF $\mathfrak{B}_2$; $\mathcal{S}_2$; OTHERWISE; $\mathcal{S}_n$
h. Stop	*stop*	STOP
i. Finish	(no equivalent)	FINISH
j. Return	*return*	RETURN
k. Procedure	I(P_i, P_i, ⁓, P_i) =: (P_o, P_o, ⁓, P_o)	$\mathcal{I}$($\mathcal{P}_i$, $\mathcal{P}_i$, ⁓, $\mathcal{P}_i$; $\mathcal{P}_o$, $\mathcal{P}_o$, ⁓, $\mathcal{P}_o$; $\mathcal{P}_r$, $\mathcal{P}_r$, ⁓, $\mathcal{P}_r$)
l. Subroutines	(no equivalent)	ENTER $\mathcal{I}$
m. Do	*do* L_1, L_2 ($S_{\rightarrow}$ → I, ⁓, $S_{\rightarrow}$ → I)	(no equivalent)

4. DECLARATIONS

a. Type	*type* (I, I, ⁓, I)	BOOLEAN $\mathcal{I}$, $\mathcal{I}$(,), $\mathcal{I}$... INTEGER $\mathcal{I}$, $\mathcal{I}$(,), $\mathcal{I}$... FLOATING $\mathcal{I}$, $\mathcal{I}$(,), $\mathcal{I}$... REAL $\mathcal{I}$, $\mathcal{I}$(,), $\mathcal{I}$...
	(no equivalent)	⟨type⟩ OTHERWISE
b. Array	*array* (I, I, ⁓, I[C:C'], I, I, ⁓)	ARRAY $\mathcal{I}_i$ ($\mathfrak{C}_i$),
	(no equivalent)	ARRAY $\mathcal{I}$; ($\mathfrak{D}_i$) = ($\mathfrak{C}_i$),
c. Functions	I(R) := E	FUNCTION $\mathcal{I}$($\mathcal{R}$) = $\mathcal{E}$
d. Comment	*comment* S;	COMMENT $\mathcal{S}$;
e. Procedure	*procedure* I(P_i) =: (P_o), I(P_i) =: (P_o), ⁓ ⁓ ⁓, I(P_i) =: (P_o) Δ; Δ; ⁓ ⁓ ⁓; Δ *begin* Σ; Σ; ⁓ ⁓ ⁓; Δ; Δ; ⁓ ⁓ ⁓; Σ; Σ *end*	PROCEDURE $\mathcal{I}$($\mathcal{P}_i$; $\mathcal{P}_o$; $\mathcal{P}_r$); BEGIN $\mathcal{S}_i$ END
f. Subroutines	(no equivalent)	SUBROUTINE $\mathcal{I}$; ($\mathcal{S}_i$)
g. Input	(no equivalent)	INPUT ($\mathcal{I}_i$ ($\mathfrak{DS}$))
h. Output	(no equivalent)	OUTPUT ($\mathcal{I}_i$ ($\mathfrak{DS}$)$_i$)
i. Format	(no equivalent)	FORMAT ($\mathcal{I}_i$ ($\mathfrak{FS}$)$_i$)

APPENDIX E

construction of machine-language programs

PROGRAMS WRITTEN IN MACHINE-LANGUAGE—whether for use as external programs, as external statements, or as library procedures—are prepared similarly, and the techniques and conventions required for their construction are listed below.

LINKAGE TO PROCEDURES

When the compiler links to any procedure, the instructions executed are:

0000 STP *aaaa*
*nn*00 BUN *aaaa*

where *aaaa* is the location of the first cell of the procedure and the value of *nn* is one less than the number of parameters given to the procedure.

Since the address fields of both the STP and the BUN instructions are the same, it is necessary that the first instruction of the procedure be a NOP. This NOP is of the form

aaaa: *bbbb* NOP *xxxx*

where *xxxx* will be replaced when the STP instruction is executed, and *bbbb* is the address where the first parameter is to be stored. Each succeeding parameter will then be stored in sequence following the address *bbbb*.

Suppose now that P_1, and P_2, ..., P_m are the addresses of those parameters which are to be given to the procedure. The compiler will in effect produce the following program:

CODE	REMARKS
0 0000 CAD P_1	
0 4400 DLB *aaaa*	
1 0000 STA 0000	$P_1 \rightarrow bbbb$
0 0000 CAD P_2	
1 0000 STA 0001	$P_2 \rightarrow bbbb + 1$
.	
.	
.	
0 0000 CAD P_{m-1}	
1 0000 STA $(m - 2)$	$P_{m-1} \rightarrow bbbb + m - 2$
0 0000 CAD P_m	$P_m \rightarrow$ rA
0 0000 STP *aaaa*	
0 *nn*00 BUN *aaaa*	$(nn = m - 1)$

Notice that the control field of the NOP instruction which heads the procedure provides the address used to determine the location in which to store parameters. The last parameter is always left in the A register. In particular, if a procedure has only one parameter, it will be found in rA.

When constructing machine-language procedures, the *bbbb* field may be either located within the procedure code itself (the control field of an instruction may be relocated—see *Relocation Conventions*) or it may be absolute. The absolute addresses available for this purpose are 0100 through 0199. A word of caution concerning the use of any of these absolute addresses for *bbbb* is necessary. The memory area 0100 through 0199 is used as a buffer area by input-output procedures. If it is possible that a procedure be executed by linkage (perhaps several times removed) from the program compiled by an OUTPUT declaration, then a conflict will result if that procedure uses the buffer area for its parameters.

PARAMETERS OF PROCEDURES

As discussed in CHAPTER VII, parameters may be considered as being either values or identifiers. Input variables or expressions are values; the procedure thus receives the actual value of the variable or expression given as a parameter. Output variables are identifiers. When an output variable is indicated, the address of that variable is given to the procedure in the 04-field.

All program-reference parameters are also identifiers; the procedure receives, in the 04-field, the address of the parameter to which reference is made.

In the case of arrays the situation is somewhat more complex. Whenever an array is written as a parameter (either as input or output) several parameters are given to the procedure. This set of parameters consists of a base address and values corresponding to each empty subscript position, which we will call m, μ_1, μ_2, ..., μ_k, respectively. The address of an element $A(n_1, n_2, ..., n_k)$ is then given by

$$m + (((...((n_1\mu_1 + n_2)\mu_2 + n_3) ... + n_k)\mu_k.$$

As an example, consider the three-dimensional array M(, , 5, , 7) which is used as a parameter to a given procedure. The parameters supplied to this procedure would be m, μ_1, μ_2 and μ_3. The procedure could then calculate the address of the element $M(n_1, n_2, 5, n_3, 7)$ by

$$m + ((n_1\mu_1 + n_2)\mu_2 + n_3)\mu_3.$$

RELOCATION CONVENTIONS

All machine-language programs are written relative to location 0000. The compiler will relocate the program so that it occupies storage starting at some other cell. The address of this cell is called the relocation base. This relocation is controlled by the sign digit of the machine-language commands.

Sign Digit of Zero, One, Two, or Three

Instructions with signs of zero, one, two, or three are not altered in any way.

Sign Digit of Four

Instructions with a sign digit of four are not allowed except for the following:

If the operation code is 00 ($sL = 62$), advance the location counter by the amount specified in the address field. This instruction behaves in a manner analogous to a pseudo-operation code in an assembler and allows the programer to reserve blocks of memory relative to the beginning of the routine.

If the operation code is 30, the compiler will insert an unconditional transfer to the statement immediately following the declarator which defined the external statement. This instruction is similar to the RETURN operation of the symbolic language and may be used any number of times within an external statement.

If the operation code is 99, the end of this machine-language program is indicated.

Sign Digit of Five or Six

Instructions with a sign digit of five or six have their address field located relative to some identifier defined within the ALGOL program. The control field is unaltered. The sign is set to one if the original sign was five, and to zero if it was six.

Sign Digit of Seven

Instructions with a sign digit of seven have their control field ($sL = 44$) relocated with respect to the first instruction of the program. The address field is not altered and the sign of the instruction is set to zero.

An instruction with a sign of seven and an operation code of 01 (NOP) is often used as the first instruction of an external or library procedure when it is desired to locate the parameters to the procedure within the routine itself.

Sign Digit of Eight

Instructions with a sign digit of eight have their address fields ($sL = 04$) relocated with respect to the first instruction of the program. The control field is not altered and the sign of the instruction is set to zero.

Sign Digit of Nine

Instructions with a sign digit of nine have their address fields relocated with respect to the first instruction of the program. The control field is not altered and the sign of the instruction is set to one.

USE OF EQUIVALENCE CARDS

It is possible in a machine-language program to refer to any identifier defined within the symbolic program, as well as to any library procedure. Every identifier to which it is desired to refer is assigned a unique two-digit equivalent, *mm*, by means of an equivalence card preceding the machine-language deck. (See APPENDIX F for the list of subroutine names). This card has the digit two in column 1, an arbitrary number of spaces, an identifier, an equal sign, and a two-digit number (leading zeros may be omitted) which is assigned to the identifier, e.g.:

COLUMNS	CONTENTS
1	2
5– 9	SIN=3

or:

1	2
7–19	SUMSQUARES=56

If the identifier is defined within a procedure, *then it must be preceded by a prefix which is the name of the procedure enclosed in parentheses.* Thus the equivalence card for label START within the procedure SIMPSON might appear as:

COLUMNS	CONTENTS
1	2
4–20	(SIMPSON)START=13

The address field of any instruction within a machine-language program which has a sign of five or six, i.e.:

$\left\{\begin{matrix}5\\6\end{matrix}\right\}$ *xxxx* OP *mm kk*

will be replaced by *kk*, plus the address of the identifier corresponding to *mm*.

For example, suppose that the instructions

```
6 0000 44 0300
6 0001 30 0301
5 0001 23 0341
```

are included within a machine-language program and that the equivalence card with the following entries:

COLUMNS	CONTENTS
1	2
4– 8	SIN=3

has preceded this program. Now assume that the compiler has assigned the cells 2856 - 2904 to the SIN routine. These instructions would then appear in the final program as:

```
0 0000 44 2856
0 0001 30 2857
1 0001 23 2897
```

If the identifier is that of an array, then the following method must be used to address a specific element within the array:

Assume that the array A has been defined previously by the declaration:

ARRAY A(μ_1, μ_2, ..., μ_k)

then the address of the element A(n_1, n_2, ..., n_k) is given by $m + (((...(n_1 \mu_2 + n_2)\mu_3 + n_3) ... + n_{k-1})\mu_k + n_k$. (Note that μ_1 is not used in the calculation).

A value of m is assigned to each array identifier by the compiler. Thus if the array M had been defined by the statement:

ARRAY M(15, 4, 6)

and the following coding was preceded by the equivalence card with the entries:

COLUMNS	CONTENTS
1	2
6– 9	M=59

the value of the element M(n_1, n_2, n_3) would be put into the A register.

LOCATION	OPERATION AND ADDRESS		REMARKS
0107	8 0000 10 0132	CAD	n_1
0108	8 0000 14 0130	MUL	μ_2
0109	0 0001 49 0010	SLT	10
0110	8 0000 12 0133	ADD	n_2
0111	8 0000 14 0131	MUL	μ_3
0112	0 0001 49 0010	SLT	10
0113	8 0000 12 0134	ADD	n_3
0114	8 0000 40 0135	STA	temp
0115	8 0000 42 0135	LDB	temp
0116	5 0000 10 5900	– CAD	m
.			
.			
.			
0103	0 0000 00 0004		μ_2
0131	0 0000 00 0006		μ_3
0132	0 0000 00 (n_1)		n_1
0133	0 0000 00 (n_2)		n_2
0134	0 0000 00 (n_3)		n_3
0135	0 0000 00 0000		temp

It has been assumed that the compiler assigned the value 3900 to m for the array M in the above example.

MAGNETIC-TAPE OPERATIONS

In order to allow the use of the MAGNETIC-TAPE FIELD SEARCH (MFS) and MAGNETIC-TAPE FIELD SCAN (MFC) commands, the following conventions have been employed:

The pseudo-operation codes 90 and 91 have been introduced for use by the programer, when referring to the MFS and MFC commands, respectively.

If the address field of the field search or scan is to be absolute, then the sign of the instruction must be four or five.

If the address field is relative to the first line of the subroutine, the sign must be eight or nine.

If the address field is relative to some identifier, the sign must be six or seven.

The signs of five, seven, and nine indicate B-modification is to be performed.

PREPARATION OF EXTERNAL PROGRAMS

As discussed in CHAPTER VII, it is necessary, when using external programs, to have an EXTERNAL declarator in the body of the symbolic program. The definition of the program itself follows the FINISH statement of the symbolic statement.

This deck must contain:

First: THE NAME CARD

'Name' cards have a two in column 1, followed by the name of the statement or procedure, a comma, and the type (INTEGER, BOOLEAN, FLOATING, or REAL). The use of a comma followed by the type is necessary only in the case of a procedure which behaves as a function, i.e., has a value associated with its name which is left in the A register.

Second: THE EQUIVALENCE CARDS

'Equivalence' cards have a two in column 1, an identifier, an equal sign, and two digits. These cards are used only when necessary to refer explicitly to other identifiers in the program.

Third: THE INSTRUCTION CARDS

'Instruction' cards define the program in machine language, and have the following format:

COLUMNS	CARD ENTRY
1 – 2	60
3	The number of instructions on this card
4 – 10	Any identification, serial numbers, etc., that the user desires
11 – 14	These columns are ignored.
15 – 25	The first instruction
26 – 36	The second instruction
37 – 47	The third instruction
48 – 58	The fourth instruction
59 – 69	The fifth instruction
70 – 80	The sixth instruction

As many of these cards as required are used. *The final card of this program must have as its last instruction the* FINISH *pseudo-operation code*: 4 0000 99 0000.

All the EXTERNAL programs declared in the symbolic program follow one after the other. Following the final external deck, another symbolic card must appear with the word FINISH on it. This defines the end of the program (symbolic statements and machine-language) to the compiler; it is in addition to the FINISH statement of the symbolic program.

EXAMPLE:
Suppose we wish to define a procedure to detect an overflow condition. In the symbolic deck, prior to the use of this procedure, we would have the declaration:

```
EXTERNAL PROCEDURE OVERFLOW (; ; L)
```

Following the FINISH statement, the following deck would appear:

CARD NO.		ENTRIES	
1	2	OVERFLOW	
2	606	0 0000 00 0000	0 0000 01 9999
			8 0410 40 0002
			0 0000 31 9999
			8 0000 42 0000
			1 0000 30 0000
			4 0000 99 0000
3	2	FINISH	

EXAMPLE:

Suppose that an external procedure is to be defined for the complex multiplication:

$(A + iB)(C + iD) \rightarrow (X + iY)$

In the symbolic program, the following declaration would appear:

```
EXTERNAL PROCEDURE CMPMULT(A ,B, C, D; X, Y); REAL
  CMPMULT;
```

After the FINISH statement would be the following deck:

CARD NO.		ENTRIES	
1	2	CMPMULT, REAL	
2	606	0 0000 01 0000	7 0020 01 9999
			8 0410 40 0017
			8 0000 41 0024
			8 0411 40 0010
			8 0000 10 0020
			8 0000 24 0022
3	606	0 0000 02 0006	8 0000 40 0024
			8 0000 11 0021
			8 0000 24 0023
			8 0000 22 0024
			0 0000 40 9999
			8 0000 10 0020
4	606	0 0000 03 0012	8 0000 24 0023
			8 0000 40 0024
			8 0000 10 0021
			8 0000 24 0022
			8 0000 22 0024
			0 0000 40 9999
5	604	0 0000 04 0018	8 0000 42 0000
			1 0000 30 0000
			4 0000 00 0005
			4 0000 99 0000

Suppose that an external procedure which will allow keyboard input of a single item of data is to be defined. The declaration

EXTERNAL PROCEDURE KEYIN (; X)

appears in the symbolic program. The card deck which follows the FINISH statement in the symbolic program would be:

CARD NO.		ENTRIES	
1	2	KEYIN	
2	606	0 0000 01 0000	0 0000 01 9999
			8 0410 40 0004
			0 0007 45 0000
			0 0000 08 0000
			0 0000 40 9999
			8 0000 42 0000
3	602	0 0000 02 0006	1 0000 30 0000
			4 0000 99 0000
4	2	FINISH	

If it was desired to use the keyboard to enter a variable ALPHA, it could be written as a statement. In the symbolic program, the declaration:

EXTERNAL STATEMENT KEYIN

would appear. Following the FINISH statement would be the deck:

CARD NO.		ENTRIES	
1	2	KEYIN	
2	2	ALPHA=33	
3	605	0 0000 00 0000	0 0007 45 0000
			0 0000 08 0000
			6 0000 40 3300
			4 0000 30 0000
			4 0000 99 0000

LIBRARY PROCEDURES

Library procedure decks are prepared in a manner identical to that for external procedures, i.e., a name card followed by equivalence cards, followed by the machine-language program. Following the final deck is again a card with the word FINISH. The method of loading these decks onto the tape is discussed in APPENDIX A, *Operating Instructions*.

EXAMPLE:

A possible library procedure for the inverse-cosine function is reproduced here as an example. This procedure uses the relation:

$$\arccos x = \arcsin(-x) + \frac{\pi}{2}$$

A reference must be made to the ARCSIN procedure. We shall assume also that cell 0047, relative to the ARCSIN procedure, is available for temporary storage and that cell 0033 contains the constant $\pi/2$.

CARD NO.		ENTRIES	
1	2	ARCCOS, REAL	
2	2	ARCSIN=21	
3	606	0 0000 01 0000	0 0000 01 9999
			6 0000 40 2147
			6 0000 11 2147
			6 0000 44 2100
			6 0000 30 2100
			6 0000 22 2133
4	603	0 0000 02 0006	8 0000 42 0000
			1 0000 30 0000
			4 0000 99 0000

THE ERROR-MESSAGE PROCEDURE

The error-message procedure controls the printing of the following types of error indication on the LINE PRINTER.

RESULT OUT OF RANGE IN $\mathcal{P}$ – $\mathcal{L}(nnnn)$
RESULT UNDEFINED FOR $\mathcal{P}$ – $\mathcal{L}(nnnn)$
RESULT ILL-DEFINED FOR $\mathcal{P}$ – $\mathcal{L}(nnnn)$
ARITHMETIC OVERFLOW – $\mathcal{L}(nnnn)$

where $\mathcal{P}$ is the label of the procedure which caused the printing. If a MONITOR statement has been given, $\mathcal{L}(nnnn)$ will also be printed, where $\mathcal{L}$ is the first five characters of the label of the last labeled statement which has been executed, and *nnnn* is the number of times this statement has been executed.

Any machine-language program may use this procedure to give error indications, provided the following conventions are adhered to:

The name $\mathcal{P}$ is in the R register, in alphanumeric form, upon entrance to the error message procedure. (In the case of arithmetic overflow, $\mathcal{P}$ is ignored.)

Upon entrance, the exit line is in the B register.

Control is transferred to locations 0000, 0007, 0014, or 0021, relative to the beginning address of the error-message procedure, to cause printing of any of the error indications listed above in their respective order.

EXAMPLE:

To illustrate the use of the error-message procedure, consider the library procedure ARCCOS. The relation

$$\cos^{-1} x = \frac{\pi}{2} - \sin^{-1} x$$

is used for calculation and hence an entry must be made to the ARCSIN routine. Also, since $\cos^{-1} x$ is undefined for $| x | > 1$, an entry must be provided to the error-message procedure.

CARD NO.		ENTRIES	REMARKS
1	2	ARCCOS, REAL	
2	2	ARCSIN=1	
3	2	ERROR=13	
4	606	0 0000 01 0000	
		0 0000 01 0000	
		8 0010 18 0012	
		8 0000 42 0000	Load B with exit line
		8 0000 41 0013	Load R with name 'ACOS'
		6 0000 34 1311	If $\| x \| > 1$, print undefined
		8 0000 40 0015	error message
5	606	0 0000 02 0006	
		8 0000 11 0015	
		6 0000 44 0100	Get $\sin^{-1}(-x)$.
		6 0000 30 0101	
		8 0000 22 0014	
		8 0000 42 0000	
		1 0000 30 0000	Exit
6	605	0 0000 03 0012	
		0 5110 00 0000	1.0
		2 4143 56 6200	'ACOS'
		0 5115 70 7963	$\pi/2$
		0 0000 00 0000	Temporary storage
		4 0000 99 0000	End of procedure

INPUT-OUTPUT PROCEDURES

External programs for specialized forms of input and output will often be employed. It is necessary in these cases to describe the coding produced by the compiler when an INPUT, OUTPUT, or FORMAT declaration is given.

The prime function of the program produced from either an INPUT or OUTPUT declaration is to form a link between those quantities which have been determined at compilation time (i.e., the addresses representing arrays, variables, expressions, etc.) and those quantities which are known only when the compiled program is executed (i.e., quantities either to be read from, or written on, various input or output media). The programs must thus be produced without reference to the routines which will use them.

When the name of an INPUT or OUTPUT declaration is given to a procedure (either a library procedure such as READ or WRITE, or some EXTERNAL procedure) it is in the program-reference field, and thus an address can be assigned to it. It is to this address that the input or output program will refer to determine an 'exit' address to which the program may transfer control. The input-output program, in turn, leaves its own return address in the B register. The A register is used to transmit the actual data. An OUTPUT declaration will put data to be transmitted to output media in the A register. An INPUT declaration will store data from the A register.

When all the relevant data have been transmitted—the input-output string having been exhausted—the sign of the A register will be loaded with the digit nine, which serves as a termination flag. *Note that no information should be transmitted until an entry has been made to the* INPUT *or* OUTPUT *declaration.* If these strings were vacuous, the A register would be loaded immediately with the termination flag, but it is necessary to enter the routine to obtain this information.

Some examples should make this clearer. Suppose we have the following INPUT declaration:

```
INPUT DATA (A, I, O(I), B)
```

Let us assume that the variables A, I, O(I), and B have been assigned cells 3701, 3702, 2008 + I, and 3703, respectively, and that the coding generated by the compiler for this INPUT declaration starts at cell 0956.

LOCATION			OPERATION	ADDRESS	REMARKS
0956	0	0000	30	0000	
0957	0	0000	42	0957	first entry
0958	0	0002	20	0956	
0959	0	0000	40	3701	second entry (A)
0960	0	0000	42	0960	
0961	0	0002	20	0956	
0962	0	0000	40	3702	third entry (I)
0963	0	0000	42	0963	
0964	0	0002	20	0956	
0965	0	0000	42	3702	fourth entry (O(I))
0966	1	0000	40	2008	
0967	0	0000	42	0967	
0968	0	0002	20	0956	
0969	0	0000	40	3703	fifth entry (B)
0970	0	0009	43	0000	termination flag
0971	0	0000	30	0956	

The following is a keyboard-input external procedure which could use this INPUT declaration.

The procedure-call in the symbolic program would be:

```
KEYIN (; ; DATA)
```

Thus only one parameter (in rA) is supplied to the external procedure. This parameter is the address of the input string DATA. In this particular case rA would contain 0 0000 01 0956.

CARD NO.		ENTRIES	
1	2	KEYIN	
2	606	0 0000 01 0000	0 0000 01 0000
			8 0410 40 0004
			8 0410 40 0005
			8 0401 26 0005
			0 0000 44 9999
			0 0000 30 9999
3	606	0 0000 02 0006	8 0009 33 0011
			8 0412 40 0005
			0 0001 45 0000
			0 0000 08 0000
			8 0000 30 0005
			8 0000 42 0000
4	602	0 0000 03 0012	1 0000 30 0000
			4 0000 99 0000

As an example of an OUTPUT declaration consider the statement:

OUTPUT DATB (A, I, X(I), B + A*2)

Assume that the compiler has assigned the cells 3095, 3096, 3097, and 2106 + I to the variables A, I, B, and X(I), respectively, and that the coding for this output statement starts at 1159.

The coding would then appear as

LOCATION			OPERATION	ADDRESS	REMARKS
1159	0	0000	30	0000	
1160	0	0000	10	3095	first entry (A)
1161	0	0000	42	1161	
1162	0	0002	20	1159	
1163	0	0000	10	3096	second entry (I)
1164	0	0000	42	1164	
1165	0	0002	20	1159	
1166	0	0000	42	3096	third entry (X(I))
1167	1	0000	10	2106	
1168	0	0000	42	1168	
1169	0	0002	20	1159	
1170	0	0000	10	3095	fourth entry (B + A*2)
1171	0	0000	24	3095	
1172	0	0000	22	3097	
1173	0	0000	42	1173	
1174	0	0002	20	1159	
1175	0	0009	33	0000	fifth entry
1176	0	0000	30	1159	termination signal

The following describes an external procedure which uses this declarator, and which will transmit the information to the SUPERVISORY PRINTER as output (in integer format).

The symbolic procedure-call might be, for example:
SPO (; ; DATB)

Thus only one parameter (in rA) would be supplied to the external procedure. This parameter is the address of the output string DATB and, in this particular case, would be 0 0000 00 1159.

CARD NO.		ENTRIES	
1	2	SPO	
2	606	0 0000 01 0000	0 0000 01 0000
			8 0410 40 0004
			8 0000 42 0004
			8 9999 21 0004
			0 0000 44 0000
			1 0000 30 0000
3	606	0 0000 02 0006	8 0009 33 0010
			8 0000 40 0012
			8 0010 09 0012
			8 0000 30 0004
			8 0000 42 0000
			1 0000 30 0000
4	602	0 0000 03 0012	4 0000 00 0001
			4 0000 99 0000

THE FORMAT DECLARATION

The compiler does not produce a program for a FORMAT declaration. It does a certain amount of preprocessing of the string language, and inserts this into the program. This preprocessing consists mainly of breaking the string up into phrases, word-by-word. The sign of the word determines the intepretation of the phrase.

PHRASE FORMAT	WORD FORMAT
@	
nnn@	
@*www*	0 *nnn* @ *www dd*
nnn@*www*	
@*www.dd*	
nnn@*www.dd*	

where @ is any alphabetic character, *nnn* is a three-digit numeric field, *www* is a three-digit numeric field, and *dd* is a two-digit numeric field.

( )	1 000 000 *aaaa*
nnn( )	1 000 *nnn aaaa*

This word corresponds to a parenthesis pair. The word appears in the phrase list at a point corresponding to the right parenthesis. The digits *nnn* are the repeat digits preceding the corresponding left parenthesis. (A zero implies an indefinite repeat.) The address of the word corresponding to the phrase following the left parenthesis is *aaaa*.

* @@@@@ * 2@@@@@

This word corresponds to five characters of an alphanumeric string lying between asterisks.

3@@@@@

This is the termination word of an alphanumeric string. One of the @'s will be an * (internal machine code: 14) showing the exact point of termination of the string. Spaces not within an alphanumeric string are ignored.

Consider the FORMAT declaration:

```
FORMAT F(5F14.8, *F(X) = *, 2(3A13, I5), W0, (5I10, P) )
```

and assume that the compiler assigns the cell 1894 to the first phrase in F. The following would then be produced. (The symbol # is used here to indicate a space):

ADDRESS	PHRASE	REMARKS
1894	0 005 46 014 08	5F14.8
1895	2 00 46 24 77 04	#F(X)
1896	3 00 33 00 14 00	#=#*
1897	0 003 41 013 00	3A13
1898	0 000 49 005 00	I5
1899	1 000 002 1897	2( )
1900	0 000 66 000 00	W0
1901	0 005 49 010 00	5I10
1902	0 000 57 000 00	P
1903	1 000 000 1901	(5I10)
1904	1 000 000 1894	F( )

Although there is a well-defined interpretation of these phrases as far as the WRITE procedure is concerned, the programer is at liberty to employ these phrases in an external program, which will interpret them in any way desired.

APPENDIX F

library procedures

In order to make alterations to the library procedures listed below, or to incorporate additional procedures in the library, it is necessary to follow the instructions given in APPENDIX A. Maintaining the library in this manner requires that the user be in possession of both the LIBRARY PROCESSOR CALLOUT deck and the appropriate library-procedure decks. The preparation of the latter is discussed in APPENDIX E, page E-5.

A description of these library procedures is given in the following pages, preceded by a specimen page showing the format of these descriptions.

The compiler tape contains the following library of procedures:

LIBRARY PROCEDURE NAMES	DESCRIPTION
FLOAT	Converts an integer to a floating-point number.
FIX	Truncates a floating-point number to an integer.
FX*FX, FL*FX, FL*FL, FX*FL	Power routines
SQRT	Square-root function
EXP	e^x
LOG	$\ln x$
SIN	$\sin x$
COS	$\cos x$
TAN	$\tan x$
ENTIRE	Greatest integer $[x]$
ARCCOS	$\cos^{-1} x$
ARCSIN	$\sin^{-1} x$
ARCTAN	$\tan^{-1} x$
SINH	$\sinh x$
COSH	$\cosh x$
TANH	$\tanh x$
ROMXX	$(1 - x^2)^{\frac{1}{2}}$
READ	Input
WRITE	Output
ERROR	Library error-message procedure (See page E-5f.)

(NAME)

FORM	The generic form used by the programer in his ALGOL program, the italicized letters indicating the input arguments for the procedures.
ARGUMENT	Specifies the type of argument required for the procedure.
RESULT	Defines the type of the result produced by the procedure.
DESCRIPTION	Outlines the operation carried out by the procedure.
ACCURACY	Self-explanatory
ERROR MESSAGE	Lists all error messages which will be printed out for the given procedure. This is shown as the error message itself, followed by two blank columns, a hyphen, the first five characters of £ (the label of the last labelled statement which has been executed), and (*nnnn*), the number of times this statement has been executed. If no MONITOR statement precedes the program, or no labelled statement has been executed, blank columns will appear in place of £ and (*nnnn*).
EXTERNAL USE	Explains conditions to be met in order to use these procedures with external machine-language procedures. The name given here is to be used on equivalence cards in external programs. (See page E-2.)
REMARKS	Self-explanatory

FLOAT

FORM	FLOAT (x)
ARGUMENT	x is integral.
RESULT	FLOAT (x) is floating-point.
DESCRIPTION	Converts the argument x into its corresponding floating-point form, as defined under ACCURACY.
ACCURACY	Exact for $\mid x \mid < 10^8$, otherwise the result is truncated to eight significant digits.
ERROR MESSAGE	None
EXTERNAL USE	x is in rA. On entry to the procedure, the exit from the subroutine must be stored in the 04 field of the first line of the procedure. Result of the procedure FLOAT (x) is in rA. Use FLOAT on equivalence cards.
REMARKS	

FIX

FORM	FIX (x)
ARGUMENT	x is floating-point.
RESULT	FIX (x) is integral.
DESCRIPTION	Truncates the argument x into its corresponding integral form. Any fractional part is lost.
ACCURACY	
ERROR MESSAGE	RESULT OUT OF RANGE IN FIX $-£(nnnn)$ This print-out will result if $\mid x \mid \geq 10^{10}$.
EXTERNAL USE	x is in rA. Use FIX on equivalence cards.
REMARKS	

POWER
ROUTINE

FXFX

FORM	*A*B*
ARGUMENTS	*A* and *B* are integers.
RESULT	The result, A^B, is an integer.
DESCRIPTION	
ACCURACY	The result is exact.
ERROR MESSAGES	RESULT OUT OF RANGE IN FXFX -£(*nnnn*) This printout will result if $\lvert A^B \rvert \geq 10^{10}$. RESULT UNDEFINED FOR FXFX -£ (*nnnn*) This printout will result if $A = 0$ and $B \leq 0$.
EXTERNAL USE	*A* is in rA; *B* is in rR. Use FX*FX on equivalence cards.
REMARKS	This routine is automatically included in the compiled program as required. *It is never called explicitly as a procedure.*

POWER
ROUTINE

FLFX

FORM	*A*B*
ARGUMENTS	*A* is floating-point; *B* is integral.
RESULT	The result, A^B, is floating-point.
DESCRIPTION	
ACCURACY	The result, A^B, has a maximum error of $\log_2 B$ in the eighth significant digit.
ERROR MESSAGES	RESULT OUT OF RANGE IN FLFX -£(*nnnn*) This printout will result if $\|A^B\| > 0.99999999 \times 10^{49}$. RESULT UNDEFINED FOR FLFX -£(*nnnn*) This printout will result if $A = 0$ and $B \leq 0$.
EXTERNAL USE	*A* is in rA; *B* is in rR. Use FL*FX on equivalence cards.
REMARKS	This routine is automatically included in the compiled program as required. *It is never called explicitly as a procedure.*

POWER
ROUTINE

FLFL

FORM	*A*B*
ARGUMENTS	*A* is floating-point; *B* is floating-point.
RESULT	The result, A^B, is floating-point.
DESCRIPTION	
ACCURACY	The error is, in general, less than 8 in the eighth significant digit. However, since the error is a function of the magnitude of A^B, the maximum error is 3 in the sixth significant digit.
ERROR MESSAGES	RESULT OUT OF RANGE IN FLFL -£(*nnnn*) This printout will result if $\|A^B\| > 0.99999999 \times 10^{49}$. RESULT UNDEFINED FOR FLFL -£(*nnnn*) This printout will result if $A = 0$ and $B \leq 0$, or if $A < 0$ and B is non-integral.
EXTERNAL USE	*A* is in rA; *B* is in rR. Use FL*FL on equivalence cards.
REMARKS	This routine is automatically included in the compiled program as required. *It is never called explicitly as a procedure.* Note that the error messages for this routine are identical to those which can occur in the power routine FXFL.

POWER
ROUTINE

FXFL

FORM	*A*B*
ARGUMENTS	*A* is integral; *B* is floating-point.
RESULT	The result, A^B, is floating-point.
DESCRIPTION	
ACCURACY	The error is, in general, less than 8 in the eighth significant digit. However, since the error is a function of the magnitude of A^B, the maximum error is 3 in the sixth significant digit.
ERROR MESSAGES	RESULT OUT OF RANGE IN FLFL -£(*nnnn*) This printout will result if $\mid A^B \mid > 0.99999999 \times 10^{49}$. RESULT UNDEFINED FOR FLFL -£(*nnnn*) This printout will result if A = 0 and $B \leq 0$, or if $A < 0$ and *B* is non-integral.
EXTERNAL USE	*A* is in rA; *B* is in rR. Use FX*FL on equivalence cards.
REMARKS	This routine is automatically included in the compiled program as required. *It is never called explicitly as a procedure.* Note that the name of the power routine FLFL occurs incorrectly in the error messages for this routine.

SQRT

FORM	SQRT (x)
ARGUMENT	x is floating-point.
RESULT	SQRT (x) is floating-point.
DESCRIPTION	SQRT (x) is the square root of x.
ACCURACY	The maximum error is 2 in the eighth significant digit.
ERROR MESSAGE	RESULT UNDEFINED FOR SQRT -£(*nnnn*) This printout will result if $x < 0$.
EXTERNAL USE	x is in rA. Use SQRT on equivalence cards.
REMARKS	

EXP

FORM	EXP (x)
ARGUMENT	x is floating-point.
RESULT	EXP (x) is floating-point.
DESCRIPTION	EXP (x) computes the exponential function e^x.
ACCURACY	Let ϵ be the error in the eighth significant digit, then for $\mid x \mid < 100$, $\epsilon \leq 3$; for $100 \leq \mid x \mid < 112.82666$, $\epsilon \leq 6$.
ERROR MESSAGE	RESULT OUT OF RANGE IN EXP -£($nnnn$) This printout will result if $x \geq 112.82666$.
EXTERNAL USE	x is in rA. Use EXP on equivalence cards.
REMARKS	

LOG

FORM	LOG (x)
ARGUMENT	x is floating-point.
RESULT	LOG (x) is floating-point.
DESCRIPTION	LOG (x) is the natural logarithm, ln x.
ACCURACY	The maximum error is 3 in the eighth significant digit
ERROR MESSAGE	RESULT UNDEFINED FOR LOG -£(*nnnn*) This printout will result if $x \leq 0$.
EXTERNAL USE	x is in rA. Use LOG on equivalence cards.
REMARKS	

SIN

FORM	SIN (x)
ARGUMENT	x is in radians, and is floating-point.
RESULT	SIN (x) is floating-point.
DESCRIPTION	SIN (x) computes the sine function.
ACCURACY	The maximum error is 4 in the eighth significant digit.
ERROR MESSAGE	RESULT ILL-DEFINED FOR SIN -£(*nnnn*) This printout will result if $\mid x \mid \geq 10^7$.
EXTERNAL USE	x is in rA. Use SIN on equivalence cards.
REMARKS	

COS

FORM	COS (x)
ARGUMENT	x is in radians, and is floating-point.
RESULT	COS (x) is floating-point.
DESCRIPTION	COS (x) computes the cosine function.
ACCURACY	The maximum error is 4 in the eighth significant digit.
ERROR MESSAGE	RESULT ILL-DEFINED FOR COS -£(*nnnn*) This printout will result if $\mid x \mid \geq 10^7$.
EXTERNAL USE	x is in rA. Use COS on equivalence cards.
REMARKS	

TAN

FORM	TAN (x)
ARGUMENT	x is in radians, and is floating-point.
RESULT	TAN (x) is floating-point.
DESCRIPTION	TAN (x) computes the tangent function.
ACCURACY	The maximum error is 9 in the eighth significant digit.
ERROR MESSAGE	RESULT UNDEFINED FOR TAN -£(*nnnn*) This printout will result if cos $x = 0$. RESULT ILL-DEFINED FOR TAN -£(*nnnn*) This printout will result if $\mid x \mid \geq 10^7$.
EXTERNAL USE	x is in rA. Use TAN on equivalence cards.
REMARKS	

ENTIRE

FORM	ENTIRE (x)
ARGUMENT	x is floating-point.
RESULT	ENTIRE (x) is floating-point.
DESCRIPTION	ENTIRE (x) computes the function normally denoted by $[x]$; it is defined to be the largest integer not greater than x.
ACCURACY	
ERROR MESSAGES	None
EXTERNAL USE	x is in rA. Use ENTIRE on equivalence cards.
REMARKS	

ARCSIN

FORM	ARCSIN (x)
ARGUMENT	x is floating-point.
RESULT	ARCSIN (x) is in radians, and is floating-point.
DESCRIPTION	ARCSIN (x) computes the inverse-sine function. The principal range is $[-\pi/2, \pi/2]$.
ACCURACY	The maximum error is 7 in the eighth significant digit.
ERROR MESSAGE	RESULT UNDEFINED FOR ASIN -£(*nnnn*) This printout will result if $\mid x \mid > 1$.
EXTERNAL USE	x is in rA. Use ARCSIN on equivalence cards.
REMARKS	

ARCCOS

FORM	ARCCOS (x)
ARGUMENT	x is floating-point.
RESULT	ARCCOS (x) is in radians, and is floating-point.
DESCRIPTION	ARCCOS (x) computes the inverse-cosine function. The principal range is $[0, \pi]$.
ACCURACY	Over most of interval [0, 1] the maximum error will be 9 in the eighth significant digit.
ERROR MESSAGE	RESULT UNDEFINED FOR ACOS -£(*nnnn*) This printout will result if $\mid x \mid > 1$.
EXTERNAL USE	x is in rA. Use ARCCOS on equivalence cards.
REMARKS	

ARCTAN

FORM	ARCTAN (x)
ARGUMENT	x is floating-point.
RESULT	ARCTAN (x) is in radians, and is floating-point.
DESCRIPTION	ARCTAN (x) computes the inverse-tangent function. The principal range is $[-\pi/2, \pi/2]$.
ACCURACY	The maximum error is 4 in the eighth significant digit.
ERROR MESSAGE	None
EXTERNAL USE	x is in rA. Use ARCTAN on equivalence cards.
REMARKS	

SINH

FORM	SINH (x)
ARGUMENT	x is floating-point.
RESULT	SINH (x) is floating-point.
DESCRIPTION	SINH (x) computes the hyperbolic-sine function.
ACCURACY	Let ϵ be the error in the eighth significant digit; then if $\mid x \mid < 100$, $\epsilon \leq 7$; if $100 \leq \mid x \mid \leq 112.82666$, $\epsilon \leq 13$.
ERROR MESSAGE	RESULT OUT OF RANGE IN SINH -£(*nnnn*) This printout will result if $\mid x \mid \geq 112.82666$.
EXTERNAL USE	x is in rA. Use SINH on equivalence cards.
REMARKS	

COSH

FORM	COSH (x)
ARGUMENT	x is floating-point.
RESULT	COSH (x) is floating-point.
DESCRIPTION	COSH (x) computes the hyperbolic-cosine function.
ACCURACY	Let ϵ be the error in the eighth significant digit; then for $\mid x \mid < 100$, $\epsilon \leq 7$. for $100 \leq \mid x \mid \leq 112.82666$, $\epsilon \leq 13$.
ERROR MESSAGE	RESULT OUT OF RANGE IN COSH -£(*nnnn*) This printout will result if $\mid x \mid \geq 112.826666$.
EXTERNAL USE	x is in rA. Use COSH on equivalence cards.
REMARKS	

TANH

FORM	TANH (x)
ARGUMENT	x is floating-point.
RESULT	TANH (x) is floating-point.
DESCRIPTION	TANH (x) calculates the hyperbolic-tangent function.
ACCURACY	Let ϵ be the error in the eighth significant digit; then for $\mid x \mid < 100$, $\epsilon \leq 10$; for $\mid x \mid \geq 100$, the result is exact.
ERROR MESSAGE	None
EXTERNAL USE	x is in rA. Use TANH on equivalence cards.
REMARKS	

ROMXX

FORM	ROMXX (x)
ARGUMENT	x is floating-point.
RESULT	ROMXX (x) is floating-point.
DESCRIPTION	ROMXX computes the function $(1 - x^2)^{\frac{1}{2}}$.
ACCURACY	The maximum error is 1 in the eighth significant digit.
ERROR MESSAGE	RESULT UNDEFINED FOR ROMXX -£(*nnnn*) This printout will result if $\mid x \mid > 1$.
EXTERNAL USE	x is in rA. Use ROMXX on equivalence cards.
REMARKS	In order to obtain accuracy for x near unity, double-precision arithmetic is used. Note that $(x^2 - 1)^{\frac{1}{2}} = x \cdot$ ROMXX $(1.0/x)$.

READ

FORMS

First form:
READ (; ; INDEC)
Second form:
READ (; s ; INDEC)

ARGUMENTS

INDEC is an identifier declared to be the name of an input-data set.

RESULT

s is a Boolean variable.

DEFINITIONS

First form:
Read in the data set INDEC from the CARD READER.
Second form:
Same as the first form, but in addition, if the word SENTINEL is encountered (other than in an alphanumeric entry), terminate the input process and set s to one. If not, set s to zero.

ACCURACY

ERROR MESSAGE

None

CALLING SEQUENCE

First form:

```
          CAD    (Address of INDEC)
          STP    READ
0  0000   BUN    READ
```

Second form:

```
          CAD    (Address of s)
          DLB    READ, sL = 44
                       nn = 00
          STA    -0
          CAD    (Address of INDEC)
          STP    READ
0  0100   BUN    READ
```

REMARKS

For further details, see CHAPTER VIII and APPENDIX E.

WRITE

FORMS

First form:
WRITE (; ; OUTSEC, FRMTDEC).
Second form:
WRITE (; ; FRMTDEC)

ARGUMENTS

OUTDEC is an identifier declared to be the name of an output-data set. FRMTDEC is an identifier declared to be the name of a format string.

RESULT

DESCRIPTIONS

First form:
Print the data set OUTDEC as output on the LINE PRINTER, the CARD PUNCH, or the SUPERVISORY PRINTER according to the format FRMTDEC.
Second form:
Print headings as output on the LINE PRINTER, the CARD PUNCH, or the SUPERVISORY PRINTER as given by the format FRMTDEC.

ACCURACY

ERROR MESSAGE

None

CALLING SEQUENCE

First form:

```
          CAD    (Address of OUTDEC)
          DLB    WRITE sL = 44
                       nn = 00
          STA    -0
          CAD    (Address of FRMTDEC)
0  0000   STP    WRITE
0  0100   BUN    WRITE
```

Second form:

```
          CAD    (Address of FRMTDEC)
          STP    WRITE
0  0000   BUN    WRITE
```

REMARKS

For further details, see CHAPTER VIII and APPENDIX E.

index

[References to definitions of terms are in italics.]

Index (continued)

Burroughs Corporation

Detroit 32, Michigan

In Canada: Burroughs Business Machines Ltd., Toronto, Ontario