800

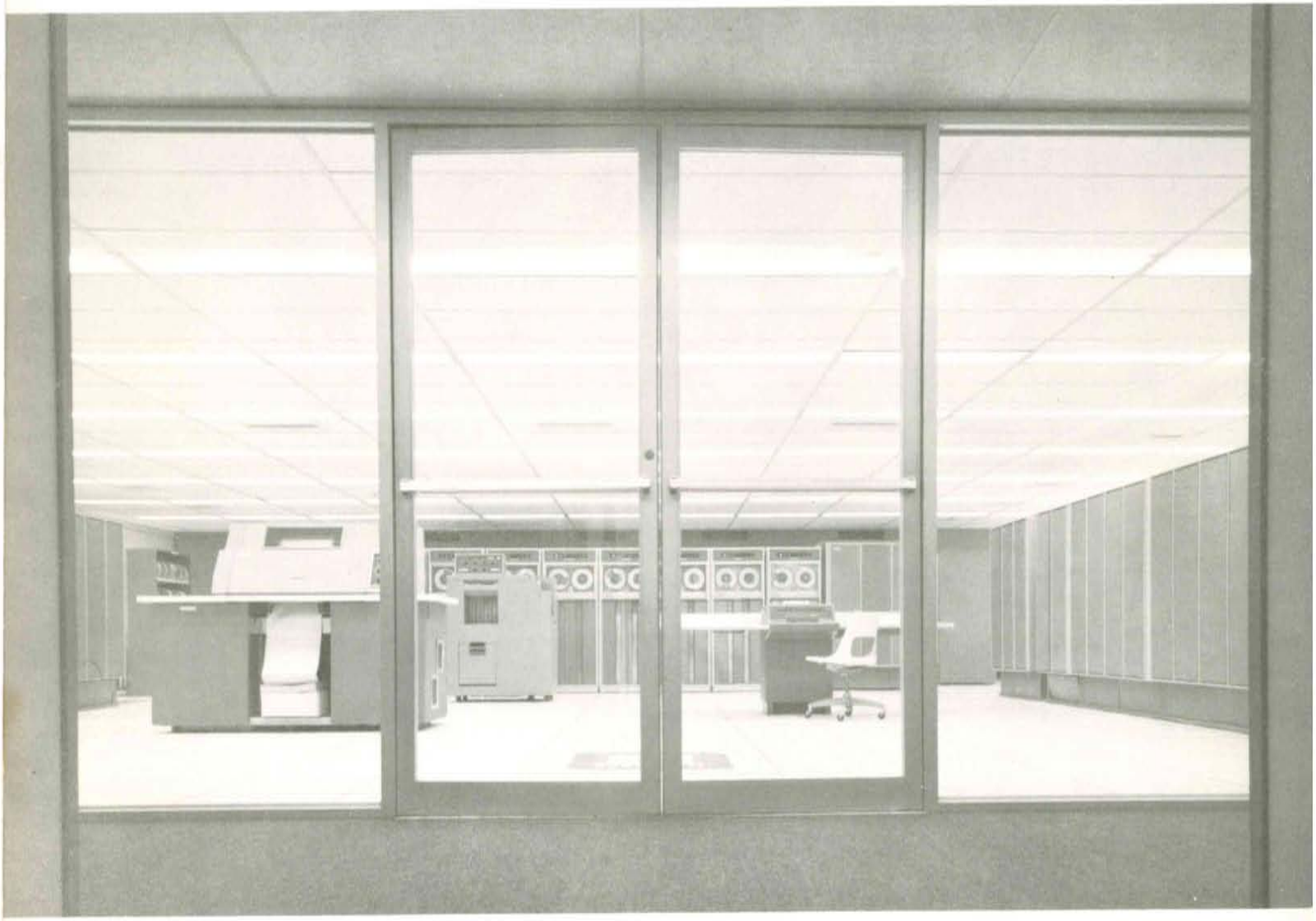
HONEYWELL

HONEYWELL

HONEYWELL 800

Transistorized Data Processing System

ALGEBRAIC COMPILER MANUAL



Litho in U.S.A.
F.I.-1156

800

HONEYWELL

ALGEBRAIC COMPILER MANUAL

PRICE $3.50

Honeywell

Electronic Data Processing

TABLE OF CONTENTS

TABLE OF CONTENTS (cont)

LIST OF ILLUSTRATIONS

FOREWORD

This manual describes the Honeywell Algebraic Compiler designed and implemented by Computer Usage Company of New York.

The manual may be thought of as consisting of three major parts. Section I gives a brief, over-all view of the Algebraic Compiler language, in terms of a representative, if short, example. This section may be of value to the reader who has no background in the subject of computer programming. Sections II through VIII make up the bulk of the manual, giving the details of the language with many examples. These sections should be useful to the beginner, as well as to the reader who is familiar with systems similar to the Algebraic Compiler. The thoroughly experienced reader may choose to turn immediately to Section IX, which presents a summary of the language without any examples. After the reader becomes proficient in the use of the Algebraic Compiler, this section will also be useful as a quick reference when it is necessary to review some topic.

PREFACE TO SECOND EDITION

This document supercedes the original edition of the Algebraic Compiler Manual (DSI-44) and should be used by all persons who are programming in the language of the compiler. The changes in the second edition, though numerous, are relatively minor and do not reflect changes in the compiler. In particular, the sample program in Section VIII has been entirely replaced by a better example and the list of language restrictions in Appendix C has been clarified.

SECTION I

INTRODUCTION TO THE HONEYWELL ALGEBRAIC COMPILER

The language of science and engineering is mathematics; the language of a computer consists of relatively elementary computer "instructions". These two languages are normally very different, requiring a translation process in order to express a mathematical procedure in the rigidly-defined format demanded by the computer. The Honeywell Algebraic Compiler greatly simplifies this translation process, by allowing the problem procedure to be expressed in a language not greatly different from ordinary mathematical notation, and by providing a separate computer program to translate from this language to the language of the computer.

The language of the Honeywell Algebraic Compiler is discussed and illustrated in Sections II through VIII, and summarized in Section IX; here, in Section I, we shall only attempt to give an over-all view of the system. It is hoped that this short introduction will make the material in the next few sections more meaningful to the reader approaching this language for the first time.

Suppose that in an engineering calculation it is necessary to compute the value of the following function, for a number of combinations of values of X and A:

$$Y = \frac{e^{AX}}{A^2 + 1} (A \sin X - \cos X) + 2.8 \cdot 10^{-4} \sqrt{10^6 + (AX)^{5.1}}$$

It is desired to set up an Algebraic Compiler program to accept 10 values of A, punched on a card. For each A, the program should compute the value of Y for all values of X between 0.1 and 2.0 inclusive, in steps of 0.1. The results are to be written onto a magnetic tape for printing at another time. Each page of the output is to consist of, first a heading line giving the value of A for the page and column headings for X and Y. The body lines will give the 20 pairs of values of X and Y.

An Algebraic Compiler program to carry out these operations is shown in Figure 1. The lines written on the coding form would be punched onto cards and processed by the Compiler, to produce a set of computer instructions which would carry out the procedure defined by the program. Thus, the compilation of the program is seen to be a separate phase from the execution of the compiled program of computer instructions. Once compiled, the program can be used with many sets of data, without recompilation. Each line

of the program shown is a separate statement, with the exception of statement number 23, which requires two lines. This statement would be punched on two cards, the second one being called a continuation card. Up to nine continuation cards are permitted.

ALGEBRAIC COMPILER STATEMENT

TITLE _____ WRITTEN BY _____ CHECKED BY _____ DATE _____ PAGE ___ OF ___

A, B, C	STATEMENT NUMBER	CONTIN	ALGEBRAIC COMPILER STATEMENT (1 / 11 / 23 / 38 / 52 / 66 / 72 / 80)
1			TITLE EXAMPLE
2			DIMENSION A(10)
3			READ 15, A
4	15		FORMAT(10F8.5)
5			DO 21 I = 1, 10
6			WRITE OUTPUT TAPE 3, 22, A(I)
7	22		FORMAT(1H1/ 2HA=, F8.5, 3B, 1HX, 13B, 1HY//)
8			X = 0.1
9	23		Y = EXPF(A(I) * X) / (A(I)**2 + 1.0) * (A(I) * SINF(X) - COSF(X))
10		X	+ 2.8E-04 * SQRTF(1E6 + (A(I) * X)**5.1)
11			WRITE OUTPUT TAPE 3, 24, X, Y
12	24		FORMAT(14B, F4.1, E20.7)
13			IF(X - 2.0) 25, 21, 21
14	25		X = X + 0.1
15			GO TO 23
16	21		CONTINUE
17			END FILE 3
18			REWIND 3
19			STOP
20			END
21	FINIS		

A	STAT. NO. DATA NAME	COMMAND CODE	A ADDRESS	B ADDRESS	C ADDRESS

Figure 1. Sample Algebraic Compiler Program

The first card of every program deck must contain a TITLE statement. This identifies the program in later phases of the operation of the Compiler system. The DIMENSION statement is related to the way in which the 10 different values of A are to be handled; the 10 values are made the 10 elements of an array named "A", and each value of A is then referred to by a subscript from 1 to 10. The DIMENSION statement written here does three things:

1. It identifies the name A as being the name of an array of variables rather than the name of a single variable;
2. The 10 in parentheses indicates that there will never be more than 10 elements in this array;
3. By the fact of having only one subscript in parentheses (in this case "10"), it indicates that this is to be a one-dimensional array.

It is also possible to have two- and three-dimensional arrays. The DIMENSION statement is called a specification statement; it gives information to the Compiler, but does not itself call for any operations to be performed. For the latter reason, it is also called a non-executable statement.

The READ statement calls for the reading of the card containing the 10 values of A. The 15 after the READ is the statement number of a FORMAT statement, which describes how the information has been punched on the card, and the type of conversion to be applied to the numbers before storing them in the computer's storage. Since the name "A" appears in a DIMENSION statement, the Compiler treats it as the name of an array without any further indication. That is, the statement "READ 15, A" calls for the entire array to be read in under control of the FORMAT statement with statement number 15. This FORMAT statement carries the following information:

1. The 10 means that there are 10 numbers, all with the same type of format and conversion required;
2. The F means that the numbers are punched without an exponent, and are to be converted to floating binary form before being stored. Floating point means simply that the numbers are stored in such a way that the machine can easily keep track of all decimal-point problems, even in a long computation involving a wide range of number sizes;
3. The 8 means that each number on the card is punched in eight columns;
4. The 5 means that there are five places after the decimal point in the numbers on the card. It is, in fact, assumed that the numbers are punched in the form ±x.xxxxx, where the x's stand for digits. In such a case, as we shall see in Section V, the 5 here is not really essential, but it does no harm.

The DO statement which comes next is one of the most powerful features of the Algebraic Compiler language; this example is a very incomplete indication of what can be done with it. The effect of this statement is: "carry out repeatedly the statements from here down through the statement with the statement number 21, the first time with I equal to 1, then with I equal to 2, etc., until the statements have been carried out with I equal to 10". In this way, we carry out the basic set of operations for each value of the parameter A. The first value of A is the one referred to by writing A(1); the same number is obtained by writing A(I), if the variable I is equal to 1. By always writing A(I) in the arithmetic statement which comes later, we get that value of A corresponding to the current value of I.

The WRITE OUTPUT TAPE statement, which follows, writes the heading information for a page. The information is placed on magnetic tape for later printing. The 3 specifies

tape unit number 3; the 22 refers to the FORMAT statement to be followed, and A(I) calls for the writing of whichever one of the A values is determined by the current value of I.

The FORMAT statement begins with a field specification "H" for Hollerith. This calls for the transmission of information directly to the output device from the FORMAT statement itself, rather than by naming a variable. The particular usage here, i.e., "1H1", is a special usage of the Hollerith field specification, however. Whenever a FORMAT statement that is used with any output statement begins with a 1H Hollerith specification, the first character following the H will be interpreted as carriage-control information and not data. The character 1 appearing here calls for spacing the paper to the head of the next form (or page) after printing the line. The 2HA= again calls for the transmission of characters from the FORMAT statement; this time the characters "A=" go directly into the output for later printing. Then the field specification F8.5 causes the number referenced by A(I) to be printed in just the same form as it was read from cards. The 3B causes three blanks to be inserted into the output, then the character X is written out as a column heading. Thirteen blanks precede the character Y as a second column heading. The two slashes cause a blank line to be inserted into the output, between the heading line and the first line of the body of the page.

The next statement is about the simplest possible example of an arithmetic statement; it causes the value of X to be 0.1. The arithmetic statement which comes next is much more complex. It begins by requesting the exponential function of the product of the current value of A(I) and X. The argument of the exponential function (i.e., AX) is enclosed in parentheses. The exponential function is obtained by writing EXPF, the pre-assigned name for this function, which is available in the Compiler. The multiplication of A(I) and X is indicated by the asterisk. The slash specifies division, in this case by $A^2 + 1$. Note that raising to a power, or exponentiation, is indicated by **. Note also that in order to show that the "1" is a floating-point number, it was necessary to write it with a decimal point. The 2 in the exponent is a fixed-point number, indicated here by the absence of a decimal point. Being fixed point means, in the Algebraic Compiler, that a number is limited to integer values and zero. Subscripts must always be fixed-point numbers, and there are also many other uses for them. The names of fixed-point variables are identified by beginning with the letter I, J, K, L, M, or N; a variable name may be from one to six characters in length. A floating-point number is identified by the fact that it begins with any letter <u>but</u> I, J, K, L, M, or N.

With this much introduction, the rest of statement number 23 should be readable. It may be noted that a floating-point constant may be written with an exponent, which is written following the letter E. When this is done, the decimal point to indicate floating point becomes

optional. Note finally that a floating-point quantity may be raised to a non-integral power.

The following WRITE OUTPUT TAPE statement puts onto tape the value of X just used and the value of Y just computed. Its FORMAT statement calls for 14 blanks to be inserted into the line, then for the first number in the "list" in the WRITE OUTPUT TAPE statement to be written out using an F4.1 field specification. X will thus print out in a four-column field with one place after the decimal point. Y is written out under control of E20.7. The E means that the number should be written with an exponent, using a total of 20 columns including blanks if necessary, and with seven places after the decimal point. For instance, the number -0.00001234567 would print out under control of E20.7 as -0.1234567E-04, which means $-0.1234567 \cdot 10^{-04}$.

The IF statement in effect asks whether X - 2.0 is less than, equal to, or greater than, zero. Depending on the answer, the next statement executed is 25 or 21. If X - 2.0 is less than zero, i. e., if X is less than 2.0, statement number 25 is executed next. This statement adds 0.1 to X, and the GO TO statement causes a transfer of control back to statement 23. If X is equal to or greater than 2.0, control transfers to the CONTINUE statement numbered 21. This is a dummy statement, used in this case because it is not possible to transfer directly back to the DO statement. By going to the CONTINUE which was named in the DO statement as the last one to be carried out repeatedly, the program "knows" to go back to the beginning of the repeated section.

After the repeated statements have been carried out 10 times, as specified in the DO statement, control will pass to the END FILE statement which will write a signal on tape that will be recognized by the printer as the end of valid information. The REWIND statement causes the output tape to be rewound. Then the STOP statement causes a short message to be typed out on the console typewriter, a message which in this case indicates that the job has been completed. The END and FINIS statements are required to properly terminate the compilation, and have no effect when the compiled program is run.

This short program has indicated some of the major concepts of the Algebraic Compiler language, without any attempt to be complete. The program uses a total of 15 different types of statements; the complete language consists of 42 types of statements. Nearly all of the types of statements used above may be used in a variety of other ways not described above.

With this much background, we may now proceed to a detailed investigation of the complete Algebraic Compiler language.

SECTION II

GENERAL PROPERTIES OF A HONEYWELL ALGEBRAIC COMPILER SOURCE PROGRAM

An Algebraic Compiler source program consists of a sequence of source statements, of which there are 42 different types. These statements are described in detail in the sections which follow, and are summarized in Section IX. A source program is written on coding sheets like the one shown in Figure 1. The source program is punched on cards, in a manner described below, and then an object program is compiled from the source program. The Compiler is itself a large computer program; the object program which it produces from a source program consists of computer instructions. The instructions of an object program "instruct" the computer to carry out operations which will produce the results specified by the statements of the source program. The instructions of the object program are compiled into a format which is determined by the characteristics of the Honeywell 800; the object program is thus said to be expressed in a machine-oriented language. The source program, on the other hand, is written in a form which much more closely resembles ordinary mathematical notation, and is said to be expressed in a problem-oriented language.

Each statement of an Algebraic Compiler source program is punched on a separate card such as the one shown in Figure 2; however, if a statement is too long to fit on one card, it can be continued on as many as nine continuation cards. The sequence of the source statements is conveyed to the Compiler only by the sequence of the statement cards.

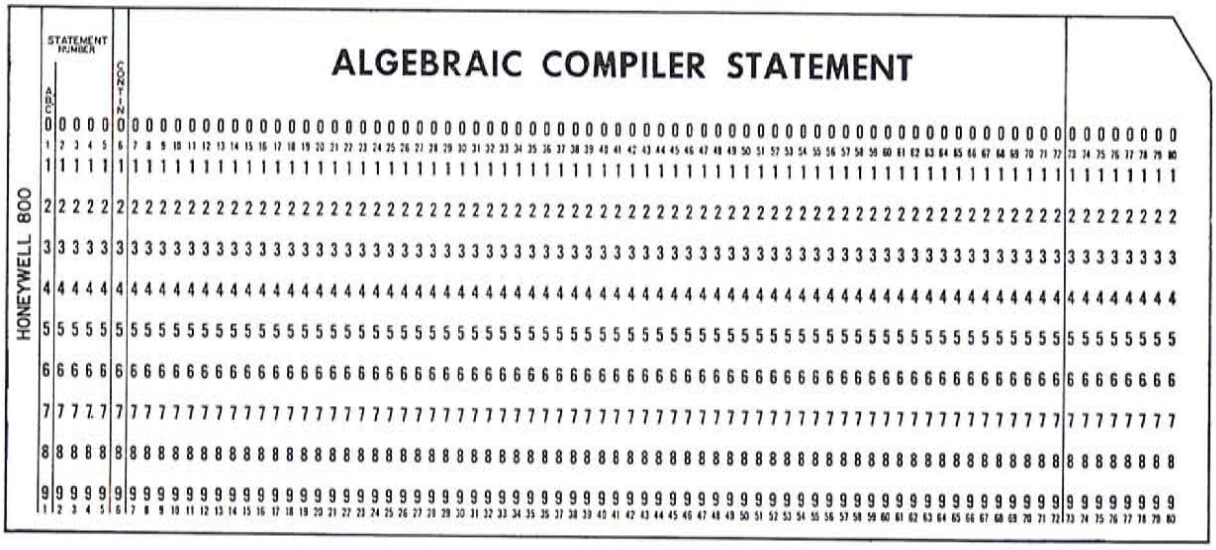


Figure 2. Algebraic Compiler Card Format

Cards which contain a "C" in column 1 are not processed by the Compiler program, and may, therefore, be used to carry comments which will appear when the source program deck is listed. Statement cards punched with a "B" in column 1 are called Boolean statement cards and are processed somewhat differently from other types of statement cards (see Section III for a discussion of Boolean statements). Statement cards with an "A" punched in column 1 are called ARGUS statement cards; such statements may be used to carry out certain Honeywell 800 instructions. Thus, it is possible to intersperse machine instructions among ordinary Algebraic Compiler statements (this subject is discussed in Section III). For all other statements, column 1 may be left blank or used as part of the statement number field.

Any number less than 32,768 may be punched in columns 1 through 5 of the first card of a statement. Such a number is called a statement number, which makes it possible to establish cross references within a source program, and facilitates the correlation of source and object programs. A statement number less than five digits in length may be punched anywhere in columns 1 through 5; leading blanks are equivalent to leading zeros. Thus, the statement number 15 may be written in any of the following ways, among others:

```
00015
15
 15
  15
   15
1    5
```

Statement numbers on cards containing an "A" or "B" in column 1 must be four or fewer digits in length. Anything at all may be punched in a card with a "C" in column 1 (comment card). Columns 2 through 5 on such a card are not interpreted as a statement number.

Statement numbers need not be written in any particular sequence, and it is not necessary to use all numbers within a range of numbers. Thus, the following would be an acceptable sequence of statement numbers:

67, 68, 49, 58, 1200, 3, 809

In short, statement numbers may be assigned in any manner whatsoever, as long as no statement number is used twice within one program. It is not necessary for all statements to have statement numbers.

Column 6 of the initial card of a statement must be left blank or punched with a zero except for certain types of statements, such as TITLE and FINIS, which will be mentioned explicitly later. Continuation cards (which must not have statement numbers) must have column 6

punched with some character other than zero. One possible method is to number the continuation cards from 1 to 9, but this is not required. Comment cards (punched with a "C" in column 1) cannot be thought of as having continuation cards. Each comment card must have a "C" in column 1, and column 6 is not interpreted as a continuation column. Continuation cards for Boolean statements must have both the "B" in column 1 and the non-zero punch in column 6. There can be no continuation card for an ARGUS statement ("A" in column 1).

The statements themselves are punched in columns 7 through 72, on both the first and continuation cards. Thus, a statement may consist of not more than 660 characters which can be punched in columns 7 through 72 of 10 cards. A table of the admissable Honeywell 800 characters appears in Appendix A. With the exceptions of the "blank" and "Hollerith" field specifications in a FORMAT statement (see Section V), blank columns in statement cards are simply ignored by the Compiler, and may be used freely to improve the readability of the source program listing.

Columns 73 through 80 of the statement cards are not processed by the Compiler, and may, if desired, be punched with identifying information.

The 42 types of statements which are available in the Algebraic Compiler system may be classified as follows:

1. The arithmetic formula statement which specifies a numerical computation. (The symbols available for referring to constants, variables, and functions, and the rules for combining of these into arithmetic formula statements are discussed in Section III.)
2. The 12 control statements which govern the flow of control in the program. (These, plus the TITLE, END, and FINIS statements, are discussed in Section IV.)
3. The 17 input and output statements which provide for input and output of data and results. (These, plus the ERASE and BUFFER statements, are discussed in Section V.)
4. The four subprogram statements which enable the programmer to define and use subprograms. (These statements and their use are discussed in Section VI.)
5. The three specification statements which provide certain necessary information about the program to the Compiler. (These statements are discussed in Section VII.)

SECTION III

CONSTANTS, VARIABLES, ARRAYS, AND ARITHMETIC STATEMENTS

The words "constant" and "variable" are used in a specialized way in the Algebraic Compiler. A constant is a number which appears in literal form in a statement. A variable is any quantity which is given a name, even through it may be used as what would be called a constant in mathematical language. Thus in the statements:

X = 21.7

Y = X + 78.2

X and Y are variables, and 21.7 and 78.2 are constants. It may be that X is never defined to be anything but 21.7; it is still called a variable.

Constants

Two kinds of constants are permitted: fixed-point and floating-point.

Fixed-Point Constants[1]

A fixed-point constant is written as from one to five decimal digits without a decimal point and without an "E" which indicates an exponent. It is thus restricted to positive and negative integer values and zero. A negative constant[1] is indicated by writing a minus sign before the constant; a positive constant may be written with or without a plus sign. The value of a fixed-point constant must be less in absolute value than $2^{15} = 32768$. Thus, the following numbers are acceptable fixed-point constants:

1, +69, -15000, 32767

Floating-Point Constants

A floating-point constant is written as no more than 16 characters with a decimal point or an "E" to indicate an exponent, or both. A decimal point may be written at the beginning or at the end, or between any two digits. Negative numbers[1] are written with a minus sign; positive numbers may be written with or without a plus sign. A floating-point number may, if desired, be written with a decimal exponent which denotes the power of 10 by which the number is to be multiplied. If this is done, the exponent is written by following the number with an "E" and then writing the exponent. Negative exponents are written with a minus sign; positive exponents may be written with or without a plus sign. The value of the floating-point constant must either lie between the approximate limits of 10^{-77} and 10^{+76} (16^{-64} and 16^{+63})

[1] Although the effect of a negative constant is easily achieved by prefixing the constant with a minus sign, negative constants as such are not generated by the Compiler.

in absolute value, or be zero. Thus, the following are all acceptable floating-point constants:

1., +1., 1.0000000

5000., 5E3, 5.E + 03

234.56, +2.3456E2, 23456.0E - 02

The three constants on each line represent the same number. The following are not acceptable floating-point constants: 6 (no decimal point or "E"); 12.98 + 2 (no "E"); 1.0 E99 (exponent too large). Constants must never be written with embedded commas.

Variables and the Names of Variables

There are four kinds of variables in the Algebraic Compiler: fixed-point; floating-point; alphanumeric; and Boolean. Fixed-point variables are named in a distinctive way, with the other three sharing one method of naming. They are named in this manner because in most problems fixed- and floating-point variables are by far the most heavily used, and because there is no question of ambiguity among the three that are named the same way.

In order to avoid ambiguity between the naming of variables and functions (see Section VI), however, it is necessary to follow two rules:

Rule 1. A variable must not be given a name which is the same as the name of a function without its final F. Thus, since there is a function named SQRTF, SQRT must not be used as a variable name.

Rule 2. A subscripted variable (see below) must not be given a name ending in F unless it is less than four characters long. Thus the name DIFF(I, J) would be in error.

Fixed-Point Variables

The name of a fixed-point variable is one to six numeric or alphabetic characters (but no special characters), of which the first is I, J, K, L, M, or N. A fixed-point variable may take on any value less than 2^{44} (approximately equal to 10^{13}) or be zero. For use in subscripts and as indexing parameters (see below), they must be less in absolute value than $2^{15} = 32768$. Acceptable names of fixed-point variables:

I, JKL, MATRIX, L1200, L1M2N3.

Unacceptable names: J123456 (too many characters); ABC (does not begin with I, J, K, L, M, or N); 5K (first character not alphabetic); $K23, J23.4 (contain special characters).

Floating-Point Variables

The name of a floating-point variable is one to six numeric or alphabetic characters (but no special characters), of which the first is alphabetic but not I, J, K, L, M, or N. A floating-point variable may take on any value which is permitted as the value of a floating-point constant, i. e., its absolute value must lie between the approximate limits of 10^{-77} and 10^{+76}, or be zero. Examples of acceptable names of floating-point variables:

BVAR, X49T, FRONT, T, R00006, CMATRX.

Unacceptable names: A123456 (too many characters); 7GROSS (first character not alphabetic); MATRIX (first character one of the excluded letters); *RED, A + B, D23.4 (contain special characters).

Alphanumeric Variables

Alphanumeric variables are named in the same way as floating-point variables. An alphanumeric variable itself consists of eight characters, with no restrictions on the characters used; specifically, "blank" is a character. For example, the following are all acceptable alphanumeric variables, although they are not acceptable names of alphanumeric variables:

12345678

ABC34.90

TODAY IS

NET*

$XXXX.XX

It must be emphasized that alphanumeric variables can only be handled in certain special ways; one obviously cannot do floating-point arithmetic on them, for instance. It must also be made clear that they are entered into the computer and stored in a manner different from the handling of any other type of variable. Alphanumeric variables can be entered into the program by use of an "A" field specification in a FORMAT statement, or may be defined as an ARGUS constant, using the "ALF" pseudo-operation (see ARGUS Manual). Furthermore, an alphanumeric variable is stored as the coded six-bit representation of each character in the eight alphanumeric character positions of a Honeywell 800 word. The number 12345.67, for instance, is stored as the alphanumeric code equivalents of the seven digits and the decimal point (period); it is not converted to the floating-point equivalent of the quantity represented by the eight characters.

Alphanumeric variables are usually manipulated with ARGUS instructions (see Section III). Other statements in which alphanumeric variables may be used, with proper planning, are:

1. As arguments in CALL statements;
2. In IF statements;
3. In the list of an input or output statement;

4. In Boolean statements;

5. In statements of the form a = b, where b is the name of an alphanumeric variable (this will cause the variable to be moved from one location to another).

The word "constant" has been defined precisely to mean a number which is written in literal form and not given a name. With this definition, there is no such thing as an alphanumeric constant in the Algebraic Compiler. That is, there is no way to write a combination of arbitrary characters and have it regarded as a literal alphanumeric value. It would clearly be impossible to do so, without providing some special mark to set off the literal constant; otherwise, there would be no way of distinguishing between, for example, DATA as the name of a floating-point variable and DATA as an alphanumeric constant. This is no real restriction. It means that every alphanumeric value must be given a name, instead of being used as a literal in a statement, as with fixed- and floating-point constants.

Boolean Variables

Boolean variables are named in the same manner as floating-point variables. A Boolean variable consists of the 48 bits of a Honeywell 800 word, a definition which is seen to include anything that might appear in a word in storage. Boolean variables are ordinarily manipulated with Boolean statements (see below), but this is partly a matter of interpretation. Since any variable can in a certain sense be regarded as a Boolean variable, presumably any Algebraic Compiler statement can be regarded as having the ability to operate on Boolean variables. As will be seen in the discussion of Boolean statements, however, what is ordinarily meant in speaking of a Boolean variable is that it is involved in a Boolean statement, so that for practical purposes we may say that Boolean variables are only operated on by Boolean statements.

Numbers to be used as Boolean variables may be entered as input by use of the "O" field specification in a FORMAT statement, or as octal numbers with the ARGUS "OCT" pseudo-operation. When the former is used, particular attention must be paid to the handling of the sign bits of the Honeywell 800 word, as discussed under the "O" field specification in a FORMAT statement (see Section V).

With the same qualifications as stated for an alphanumeric variable, there is no such thing as a Boolean constant in the Algebraic Compiler.

Subscripted Variables

It is often necessary or convenient to deal with one-, two-, or three-dimensional arrays. A one-dimensional array corresponds to a vector and a two-dimensional array to a matrix, but

there are also many circumstances in which arrays are convenient other than in working with vectors and matrices. Individual elements in arrays are referred to by using subscripted variables, which have the name of fixed- or floating-point variables (whichever applies) followed by parentheses enclosing the one, two, or three subscripts. These are separated by commas if there is more than one. A subscript is a fixed-point quantity, the value of which determines the element in the array to which reference is made. For example, in working with an array named COST, the second element in the third row could be referred to by COST(3, 2). This is an example in which both subscripts are fixed-point constants. A subscript can be expressed in any of the following forms:

1. A fixed-point constant;
2. A fixed-point variable;
3. A fixed-point variable plus or minus a fixed-point constant;
4. A fixed-point constant times a fixed-point variable;
5. A fixed-point constant times a fixed-point variable, plus or minus a fixed-point constant.

If ADATA is the name of a three-dimensional array, the following are examples of acceptable subscripts:

```
ADATA(5, I, J + 2)
ADATA(2 * K, J - 4, 3 * I + 8)
```

The following are examples of unacceptable subscripts:

ADATA(2.0, I * J, K * 2)	(2.0 is a floating-point constant. The product of two fixed-point variables such as I * J is not permitted. Should be 2 * K, not K * 2.)
ADATA(5 + M, I + K, B)	(Should be M + 5, not 5 + M. The sum of two fixed-point variables is not permitted. B is a floating-point variable and may not be used as a subscript.)
ADATA(+2, -N, +J + 9)	(Leading signs are not permitted.)

A variable in a subscript must not itself be subscripted. A variable which appears in subscripted form must appear in a DIMENSION statement to specify to the Compiler the number of dimensions in the array and the maximum value of each subscript. A subscript variable must always be written with the same number of subscripts as in its DIMENSION statement. (See Section VII for a complete description of the DIMENSION statement.) The value of a subscript must be greater than zero, since there is no zeroth element. The first element of an array is the one corresponding to the subscripts (1) or (1, 1) or (1, 1, 1). Furthermore, the value of a subscript must not be greater than the corresponding maximum size given in the DIMENSION statement.

For instance, in ADATA(I + 3, 2 * K, K - 8), I may be less than or equal to 1 as long as I + 3 exceeds zero; 2 * K must not exceed the maximum size for the second subscript; K may exceed the maximum size given for the third subscript as long as K - 8 does not.

An array is stored with the element corresponding to the subscript (1), (1, 1), or (1, 1, 1) in the lowest-numbered location and the other elements in consecutive ascending locations. Two- and three-dimensional arrays are stored in consecutive locations in such a way that their first subscript (from the left) varies most rapidly and their last subscript varies least rapidly. For instance, a 2 x 2 x 3 array with the name A would be stored, with the first element in the lowest-numbered location, as follows:

A(1, 1, 1), A(2, 1, 1), A(1, 2, 1), A(2, 2, 1),
A(1, 1, 2), A(2, 1, 2), A(1, 2, 2), A(2, 2, 2),
A(1, 1, 3), A(2, 1, 3), A(1, 2, 3), A(2, 2, 3).

Expressions

The word expression is used in a special sense in the Algebraic Compiler to indicate any of a variety of allowable combinations of constants, variables, and functions. Every expression is either of the fixed-point or the floating-point mode, depending on whether the value of the expression is a fixed-point or floating-point number. (See Section VI for a discussion of functions.) Repeated application of the following set of rules will lead to any permissible expression, with the exception of Boolean expressions which are discussed separately. Several of these rules relate to the mode of an expression.

Rule 1. Any fixed-point or floating-point constant, variable, or subscripted variable, is itself considered to be an expression.

Therefore, when it is stated below that the right-hand side of an arithmetic statement may be any expression, single constants and variables are included. Thus, the following are all expressions:

A, 789, 34.987E - 4, DATA(7, 3, 1), VECT(K).

Rule 2. There are a limited number of cases in which it is permissible to have expressions of mixed mode, i. e., containing both fixed- and floating-point quantities:

1. A floating-point quantity can appear in a fixed-point expression only as an argument of a function;
2. A fixed-point quantity can appear in a floating-point expression only as an argument of a function, or as an exponent, or as a subscript.

Thus, the following are all expressions:

XINTF(A), FLOATF(NODE), A ** 2, CROSS ** N, VOLTS(8), WATTS(I, J, K).

The character combination ** is used to denote exponentiation. XINTF(A) is the name of a defined function with a floating-point argument whose value is to be fixed point (see Section VI), while FLOATF(NODE) is the name of a defined function with a fixed-point argument whose value is to be a floating-point quantity.

Rule 3. A function is an expression, if expressions of the correct modes are specified as its arguments. The mode of the function considered as an expression is the same as the mode of the value determined by the function. Examples:

ABSF(X), XABSF(I), SINF(THETA). (See Figure 6)

Rule 4. If E is any expression, and if its first character is not + or -, then +E and -E are expressions of the same mode as E. Examples:

+TEMP, -I, -78, +64.77, -CURR ** 2.

Rule 5. If E is any expression, then (E) is an expression of the same mode as E. Thus, the following are all expressions:

(RHO), ((RHO)), (((RHO))), (+9.0).

This rule is used to group expressions as in normal mathematical practice, and is demonstrated in the second example under Rule 6.

Rule 6. If E and F are any expressions of the same mode, and if the first character of F is not + or -, then the following are all expressions of the same mode as E and F:

E + F

E - F

E * F

E / F

The characters +, -, *, and / are used to denote addition, subtraction, multiplication, and division, respectively. As in ordinary algebra, we must distinguish between + used to denote addition and + used to denote a positive number (similarly with the - sign). The usual algebraic rules hold; e.g., E + (-F) and E - (+F) are equivalent, but because of Rule 4, the parentheses are essential. Parentheses are used in arithmetic expressions very much as in ordinary algebra. For instance, A - B + C and

A - (B + C) are both legitimate expressions, but they do not mean the same thing because of Rule 5. Parentheses never replace the * required to indicate multiplication. Examples:

I + 2, X - Y, AVAL / 2.0, B * C, RHO + SIGMA - TAU,
2.0 * (U + V), SQRTF(AREA) / 2.0, 2 * K - 6, (M * N) / 2

Rule 7. If E and F are expressions, if F is a floating-point expression only if E is, if neither E nor F is of the form A ** B, and if the first character of F is not + or -, then E ** F is an expression of the same mode as E. The character combination ** is used to denote exponentiation. Examples:

A ** 3.5, X ** Y, 6 ** (I - 2), (X -Y) ** XABSF(KK -LL).

The following examples all violate some one or more of the rules:

A + M	This is a mixed expression (Rule 2).
I ** 1.5	A fixed-point number cannot be raised to a floating-point power (Rule 7).
- +A + B	Violates Rule 4. Must be written as -(+A) + B.
A ** B ** C	Violates Rule 7. Must be written as A ** (B ** C) or (A ** B) ** C, whichever is intended.
X * - Y	Violates Rule 6, which states implicitly that two operation symbols must not be written consecutively. Rewrite as X * (-Y) or -X * Y.
SQRTF(I + J)	Violates Rule 3. This (see Figure 6) function requires a floating-point argument.

Hierarchy of Operations

Expressions often arise which would be ambiguous in the absence of some rules to define the order in which operations are performed. For instance, does A / B * C mean $\frac{A}{B} \cdot C$ or $\frac{A}{B \cdot C}$? (As seen in the following rules, the former would be the meaning.) Three rules govern such situations.

Rule 1. Parentheses override everything else. If the expression A * (B + C) + D / (E * F) is written, then the meaning is $A \cdot (B + C) + \frac{D}{E \cdot F}$, regardless of the two rules below.

Rule 2. In the absence of parentheses, the hierarchy of operations is:

1. Exponentiation;
2. Multiplication and division;
3. Addition and subtraction.

In other words, in the absence of parentheses, all exponentiations are performed first, then all multiplications and divisions, then all additions and subtractions. Thus, the following two expressions are equivalent:

1. A * B + C / D - E ** F
2. (A * B) + (C / D) - (E ** F)

Rule 3. Expressions in which parentheses are omitted from a sequence of consecutive additions and subtractions, or a sequence of consecutive multiplications and divisions, are treated as though there were parentheses grouped from the left. With multiplication and division this rule avoids what is considered to be an ambiguity in mathematics. In the example cited above, A / B * C denotes (A / B) · C.

Another example, A * B / C / D * E * F would mean (((((A * B) / C) / D) * E) * F), which could also be rewritten as (A * B * E * F) / (C * D). It is a good procedure to put in extra parentheses when in doubt.

Arithmetic Statements

An arithmetic statement is of the general form a = b, where a is a subscripted or non-subscripted variable, and b is an expression. A constant must never be written on the left-hand side of an arithmetic statement. An arithmetic statement closely resembles a conventional algebraic formula. The important difference is that the = sign is not used here in the sense of "is equivalent to", but rather is used to mean "the value defined by the expression b replaces the previous value of a". Examples of arithmetic statements:

ALGEBRAIC COMPILER STATEMENT

TITLE _____ WRITTEN BY _____ CHECKED BY _____ DATE _____ PAGE ___ OF ___

A, B, C	STATEMENT NUMBER	CONTIN	ALGEBRAIC COMPILER STATEMENT (1 / 6 / 11 / 23 / 38 / 52 / 66 / 72 / 80)
			A = 1.0
			JINDEX = 2
			X = 12.*Y + (Z-2.)**2
			ROOT1 = (-B+SQRTF(B**2 -4.*A*C))/(2.*A)
			ADATA(1,2,4)=BDATA(2,2,2)
			I=I+1

The last statement, which in itself is a mathematical inaccuracy, is a perfectly legitimate illustration of the specialized usage of the = sign in the arithmetic statement. This example indicates that the value of the fixed-point variable is replaced by the value I + 1.

The result of a calculation defined by an arithmetic statement will be in floating-point form if the variable on the left-hand side of the = sign is named as a floating-point variable and is in fixed-point form if the variable on the left is named as a fixed-point variable. If the variable on the left is named as a floating-point variable and the expression on the right is fixed point, the result is first computed using fixed-point arithmetic, and then converted to floating-point form. If the variable on the left is named as a fixed-point variable and the expression on the right is floating point, the result is first computed using floating-point arithmetic, then truncated and converted to a fixed-point integer. Truncate, as used here, means to discard any fractional part of the result without rounding. Thus, the statements:

```
I = 1.6 * 3.
```

would give 4 for the value of I, not 5. The statement:

```
J = 8. / 3.
```

would give 2 for the value of J, not 3.

The following examples illustrate how arithmetic statements can be written and used.

Given two points in a plane represented by the Cartesian coordinates (X1, Y1) and (X2, Y2), the distance D between the two points is given by:

$$D = \sqrt{(X2 - X1)^2 + (Y2 - Y1)^2}$$

which can be written as:

$$D = ((X2 - X1)^2 + (Y2 - Y1)^2)^{\frac{1}{2}}$$

then the following arithmetic statement would result in the computation of D:

ALGEBRAIC COMPILER STATEMENT

TITLE _____ WRITTEN BY _____ CHECKED BY _____ DATE _____ PAGE ___ OF ___

A, B, C	STATEMENT NUMBER	CONTIN	ALGEBRAIC COMPILER STATEMENT
1		6	11 ... 23 ... 38 ... 52 ... 66 ... 72 ... 80
			D = ((X2-X1)**2 + (Y2-Y1)**2)**0.5

All of the parentheses here are essential.

The area of a triangle, whose sides are of length A, B, and C, is given by:

$$AREA = \sqrt{S(S - A)\ (S - B)\ (S - C)}$$

where:

$$S = \tfrac{1}{2}(A + B + C).$$

The area would be computed by the following two statements:

ALGEBRAIC COMPILER STATEMENT

TITLE _____ WRITTEN BY _____ CHECKED BY _____ DATE _____ PAGE ___ OF ___

A, B, C	STATEMENT NUMBER	CONTINUATION	ALGEBRAIC COMPILER STATEMENT
1			S = 0.5 * (A + B + C)
2			AREA = (S*(S-A)*(S-B)*(S-C))**0.5

The scalar product of two vectors with components (x_1, x_2, x_3) and (y_1, y_2, y_3) is defined to be $x_1y_1 + x_2y_2 + x_3y_3$. Given the two one-dimensional arrays with names XVECT and YVECT, one way to compute the scalar product, which we shall call SCPROD, would be through the use of the following statement:

ALGEBRAIC COMPILER STATEMENT

TITLE _____ WRITTEN BY _____ CHECKED BY _____ DATE _____ PAGE ___ OF ___

A, B, C	STATEMENT NUMBER	CONTINUATION	ALGEBRAIC COMPILER STATEMENT
1			SCPROD = XVECT(1)*YVECT(1) + XVECT(2)*YVECT(2) + XVECT(3)*YVECT(3)

Another way to accomplish the same end will illustrate the use of subscripts, although in this case it would be more trouble than the method shown above. The value of SCPROD is first set to zero, and then the scalar product is accumulated in SCPROD, one product at a time, SCPROD does not contain the complete scalar product until the program has been completed.

ALGEBRAIC COMPILER STATEMENT

TITLE _____ WRITTEN BY _____ CHECKED BY _____ DATE _____ PAGE ___ OF ___

A, B, C	STATEMENT NUMBER	CONTINUATION	ALGEBRAIC COMPILER STATEMENT
1			SCPROD = 0.0
2			I=1
3			SCPROD = SCPROD + XVECT(I)*YVECT(I)
4			I=2
5			SCPROD = SCPROD + XVECT(I)*YVECT(I)
6			I=3
7			SCPROD = SCPROD + XVECT(I)*YVECT(I)

It should be noted that three of the statements above are exactly the same, with the value of the subscript being the only thing that changes. This method, although it will work, is unnecessarily cumbersome. Section IV specifies the use of a DO loop to carry out this computation in a parallel but simpler manner.

Integer Arithmetic

Arithmetic statements involving computation on fixed-point quantities are carried out using integer arithmetic. The important consideration here is that if a division produces a result which is not an integer, the fractional part is not rounded but truncated (dropped). Thus, the floating-point division 8. / 3. would give 2.666.... , but the fixed-point division 8 / 3 would give 2, the largest integer not greater than 8 / 3.

Integer arithmetic may produce a result which is greater than 32768, but it must be less than or equal to 2^{44}, which is approximately 10^{13}. If an integer equal to or larger than 32768 is used as a subscript or as an indexing parameter in a DO statement, only the rightmost 15 bits will be used, i. e., it will be reduced modulo 2^{15}. On output, all 44 bits (and sign) can be written out.

Boolean Statements

Sometimes it is desirable to use Boolean algebra, either to perform logical operations or, in certain circumstances, to obtain the effect of masking by using appropriate Boolean operations. Wherever the operations are defined, it is possible to specify Boolean algebra by placing a "B" in column 1. This procedure may be followed in ordinary arithmetic statements, IF statements, in function definitions, and with the arguments of a CALL statement.

The elements on which Boolean operations are performed must have the names of floating-point variables, i. e., the names do not begin with I, J, K, L, M, or N. Boolean operations on literal constants are excluded. The variables may have been defined by any convenient method, including definition by an "O" format or as an ARGUS constant.

Four Boolean operations are provided in the Compiler. Logical addition, indicated by +, is the inclusive OR function. Each of the 48 bit positions of the two words is treated separately; in each position, the logical sum is 1 if either or both bits are 1, and zero only if both are zero. As an example:

```
OR      00111111000
        10101010101
        ----------
        10111111101
```

Logical multiplication, indicated by *, is the AND function. In each bit position, the result is 1 if and only if both bits in the corresponding position in the two words are 1. As an example:

```
AND          00111111000
             10101010101
             ----------
             00101010000
```

The exclusive OR function of two variables is specified in the Compiler by the use of the Boolean function EXCLORF. The two arguments of the function must be stated. The result in each bit position is 1 if either bit in the corresponding position in the two words is 1, and zero if both are zero or if both are 1. As an example:

```
Exclusive OR 00111111000
             10101010101
             ----------
             10010101101
```

Complementation, indicated by -, applies to one variable or expression. In each bit position, a 1 is replaced by zero and zero by 1. Complementation must not be confused in any way with subtraction. Subtraction and division are not defined in Boolean algebra, and cannot be. An expression such as A + (-B) ("form the logical sum of A and the complement of B") is correct but A - B is not because it attempts to apply complementation as though it were a relationship or operation involving two expressions, which it is not.

It should be made clear that the Boolean operations may be applied to expressions involving many variables, but enough parentheses must be used to avoid ambiguity, especially with complementation. Thus the expressions A + (-B) + C and A + (- (B + C)) are both legitimate, although they do not mean the same thing. One further example of an acceptable Boolean expression is:

```
- ((A + B) * (C + D))
```

If a statement is specified as Boolean, the specification will be applied to all the algebra in a statement. There is no way at all of mixing Boolean and "ordinary" algebra in one statement. This perhaps might most easily be overlooked in the use of Boolean algebra on the arguments of a CALL statement. Usage of Boolean algebra is permissible in a CALL statement, but only if one wishes all the algebra in the CALL to be Boolean. Note in this connection, however, that one might have a situation where it is convenient to specify Boolean algebra on the CALL arguments although none or only part of the algebra in the subroutine involved is Boolean. In short, each statement stands by itself. All the algebra within one statement must be one type or the other.

To give some indication of how these operations might be used, consider the following example. Suppose that a routine is being programmed to operate efficiently on sparse square matrices having elements a_{ij}, i. e., consisting mostly of zeros, of up to order 48. It has been decided to set up 48 control words in a one-dimensional array called CNTRL which will specify which of the elements are non-zero, according to the following scheme. If the a_{11} element of the matrix is non-zero, bit 1 (the leftmost bit of CNTRL(1)) is to be a 1; if the a_{12} element is non-zero, bit 2 of CNTRL(1) is to be 1; and similarly for the other bits on the first control word and the elements of the first row. The elements of the second row are described by the bits of CNTRL(2), etc. Stated concisely, bit J of CNTRL(I) is a 1 if the a_{ij} element is non-zero and zero if it is zero. Now if the convention is adopted that the elements are stored in column order, it is only necessary to store the non-zero elements and the control words, instead of storing the entire matrix. The latter scheme, under the sparseness hypothesis, would involve much wasted tape and core space.

Suppose now that all this has been done, and that it is necessary to know if any elements in the first column are non-zero. This requires inspecting bit 1 of as many control words as the order of the matrix, which we will call N; if any one or more of these bits is 1, then we wish to take a different path through the computation than if they are all zero. Suppose now that there is in storage a word called BIT1, which consists of a 1 in bit 1 and zeros in all other bit positions. The following Compiler program will jump to statement 100 in the all-zero case and to 200 if there are any non-zero elements.

ALGEBRAIC COMPILER STATEMENT

TITLE _____ WRITTEN BY _____ CHECKED BY _____ DATE _____ PAGE ___ OF ___

	A, B, C	STATEMENT NUMBER	CONTIN	ALGEBRAIC COMPILER STATEMENT
1	B			TEST = EXCLORF(BIT1,BIT1)
2				
3				DO 20 I = 1,N
4				
5	B	20		TEST = TEST + CNTRL(1)
6				
7	B			IF (TEST * BIT1) 200, 100, 200

The first statement simply clears to 48 zeros the temporary storage location TEST. The result might be called a "Boolean zero", in distinction to an ordinary plus zero which has ones in the sign bits. The DO loop is to be executed once for each control word, i. e., N times.

Statement 20, which is indicated as a Boolean statement, forms the logical sum of the bits in each position of all of the control words. The (Boolean) IF statement forms the logical product of the result of the DO loop and the mask, which is a word that has been set up to be all zeros except in bit 1. The IF statement then asks, in effect, whether or not the result of this is zero. By making the IF a Boolean statement, we specify a test for a word of 48 zero bits.

ARGUS Statements

It is expected that most problems on which the Algebraic Compiler is used can be satisfactorily solved using only the various statements of the Compiler language, but there will undoubtedly be some tasks which can be handled more easily or simply using a mixture of Compiler statements and ARGUS instructions. This flexibility is permitted in the Algebraic Compiler system, which allows one to intersperse instructions written in the ARGUS language among ordinary Compiler statements. The ARGUS instructions may be written on the Compiler coding form as desired in the compiled program, by placing an "A" in column 1 of the statement line of the Compiler coding form (see Figure 3).

The rest of the ARGUS instruction format is:

Columns 2 - 5: Statement Number or Blank
Columns 11 - 22: Operation Code
Columns 24 - 37: A Address
Columns 38 - 51: B Address
Columns 52 - 65: C Address

The discussion which follows assumes a rudimentary knowledge of the ARGUS system.

The allowable types of addresses used in ARGUS statements are limited to names of floating-point variables, ARGUS constants, literal floating-point constants without a sign, statement numbers or binary counts according to the following table:

Type of Operation	A Address	B Address	C Address
Arithmetic (floating binary)	General	General	Variable
Logical	Symbol	Symbol	Variable
Comparison	General	General	Statement Number
TS	General or Inactive	Variable or Inactive	Statement Number
TX	General		Variable
Shift	Symbol	Binary Count	Variable
Print	General	Inactive	Statement Number or Inactive

The result location may be general in all cases and the compiler will allow it, but it is advisable that the above rules be followed. In the above table the following definitions apply:

Variable - name of a floating-point variable.

General - includes name of floating-point variable and ARGUS constant and literal floating-point constants.

Symbol - includes name of floating-point variable and ARGUS constant.

The portions of the ARGUS vocabulary which may be used include the three ARGUS constant pseudo-instructions ALF, OCT, and FLBIN, with the restriction that these must appear with only one entry per statement line. In addition to these data entry instructions, the Algebraic Compiler permits the use of:

BA	Binary Add
BS	Binary Subtract
BM	Binary Multiply
BD	Binary Divide
WA	Word Add
WD	Word Difference
HA	Half Add
SM	Superimpose
SS	Substitute
EX	Extract
TX	Transfer
TS	Transfer and Sequence Change
NN	Inequality Comparison, Numeric
NA	Inequality Comparison, Alphabetic
LN	Less Than or Equal Comparison, Numeric
LA	Less Than or Equal Comparison, Alphabetic
SPS	Shift Preserving Sign and Substitute
SPE	Shift Preserving Sign
SWS	Shift Word and Substitute
SWE	Shift Word and Extract
PRA	Print Alpha
PRD	Print Decimal
PRO	Print Octal
FBA	Floating Binary Add
FBS	Floating Binary Subtract
FBM	Floating Binary Multiply
FBD	Floating Binary Divide

FLN Normalized Less Than Comparison

FFN Fixed to Floating Normalize

FNN Floating Inequality Comparison

All other ARGUS instructions are specifically excluded from the set of permissible instructions in the Algebraic compiler.

Further, ARGUS instructions may not use the cosequence mode, simulator instructions, or masking. No Compiler functions may be addressed.

The following example may help to clarify the use of the interspersed ARGUS instructions in the Algebraic Compiler. Assume that a part of the input to a problem is a list of eight-letter names which are to be printed out in alphabetic sequence; associated with each of them is a four-digit fixed-point number. Assume further that the names and the numbers have already been read in, and are stored as arrays in ABC(I) and NUMB(I), respectively. The task is to sort the two arrays into sequence on the alphabetical contents of the first array, and then print out on the typewriter the first name in alphabetic sequence and the corresponding number. This number is to be printed in octal. The size of the array is given by the fixed-point variable "NO".

The first problem is that the Compiler was not intended to handle such things as sorting alphabetics. One solution is to employ ARGUS statements and use the LA instruction for making the required comparisons. The second problem is how to do the sorting. The method used here is not necessarily the best in all cases, but it will be adequate; selection sorting. With this method, a comparison is made between first and second, first and third, etc., and finally the first and Nth words, exchanging each pair that is out of sequence; this "selects" the smallest word in the list and moves it to the top. The process is repeated on the second and third, second and fourth, etc., and finally the second and Nth, to get the next larger word, etc., etc., until finally a comparison and exchange, if necessary, is made between the (N - 1)st and Nth words.

Each time an exchange is necessary, not only the names, but also the numbers must be exchanged. The exchanging of the names may be done either in Compiler terms or in ARGUS, since they look like floating-point variables, but the exchanging of the numbers will have to be done in Compiler terms since they are fixed point. An alternative method would be to convert to floating point beforehand, but this clearly would not be worth the effort. In carrying out the exchanging, two temporary locations are needed. These will be called TEMP and NTEMP; there must be two because of working with both floating and fixed variables. The program could be as shown in Figure 3.

The first three statements are required to define the symbols used and to allocate memory locations to them. NM1 and IP1 had to be set up because the only expressions that may be used in a DO statement are fixed-point variables or constants. The first ARGUS statement makes an alphabetic less-or-equal comparison between two names in the array, and jumps around the interchange if they are already in sequence. The two DO loops carry out the sequence of tests and interchanges described above. The two print instructions print the smallest name and the corresponding number.

The use of ARGUS arithmetic statements is not illustrated here since their use should be relatively straightforward to anyone with a basic understanding of ARGUS. A situation where the interspersing of ARGUS statements may be quite useful is in operating on "packed" words, i. e., words containing more than one quantity, as is frequently done in tape operations in data processing. The direct use of the shift instructions, which of course has no direct counterpart in the Algebraic Compiler language itself, would then be required.

ALGEBRAIC COMPILER STATEMENT

TITLE _____ WRITTEN BY _____ CHECKED BY _____ DATE _____ PAGE ___ OF ___

	A, B, C	STATEMENT NUMBER	CONTIN	ALGEBRAIC COMPILER STATEMENT				
	1		6	11	23	38	52	66 72 80
1		TITLE		SORT				
2				DIMENSION ABC(1000), NUMB(100)				
3				TEMP = 0.0				
4				NTEMP = 0				
5				NM1 = NO - 1				
6				DO 20 I = 1, NM1				
7				IP1 = I + 1				
8				DO 20 J = IP1, NO				
9	A			LA	ABC(I)	ABC(J)	20	
10	A			TX	ABC(I)		TEMP	
11	A			TX	ABC(J)		ABC(I)	
12	A			TX	TEMP		ABC(J)	
13				NTEMP = NUMB(I)				
14				NUMB(I) = NUMB(J)				
15				NUMB(J) = NTEMP				
16		20		CONTINUE				
17	A			PRA	ABC(1)	-	-	
18	A			PRO	NUMB(1)	-	-	
19				STOP				
20				END				
	A	STAT. NO. DATA NAME		COMMAND CODE	A ADDRESS	B ADDRESS	C ADDRESS	

Figure 3. Alphabetic Sort Example

SECTION IV
CONTROL STATEMENTS

In any meaningful problem to be carried out using the Algebraic Compiler, it may be necessary to alter the flow of control of the statements from the one-after-the-other sequence which is followed in the absence of any control statements. Often there are alternative parts of the program which must be executed, depending on the data. Frequently there are sections of the program which must be repeated, either to operate successively on new sets of data or to carry out some iterative process. The Algebraic Compiler provides a number of statements for handling such situations.

In setting up any program, it is necessary to observe the following rules, which are to some extent self-evident, but which can lead to serious difficulties if not observed.

Rule 1. Every program must terminate on a STOP or PAUSE statement. This does not mean that a STOP or PAUSE must be the last statement physically (indeed this is impossible since END must be the last statement of every program), but that every program must be able to reach a STOP or PAUSE when the computation is completed.

Rule 2. Some path of control must be able to reach every executable statement in a program. This does not mean that every statement must be executed each time a given program is run, but that there must not be any executable statements which could never be executed under any circumstances.

Rule 3. All transfers of control must be to executable statements, never to FORMAT, DIMENSION, EQUIVALENCE, COMMON, END, or ARGUS data entry constants.

Unconditional GO TO Statement — GO TO n

This statement is used when it is desired simply to break out of the sequential execution of statements without making the transfer of control conditional. Control is unconditionally transferred to the statement with the statement number n.

Computed GO TO Statement — GO TO $(n_1, n_2, \ldots, n_m)$, i

This statement is used when it is desired to transfer control to one of a number of statements, depending on the current value of some fixed-point variable. In this statement, i must

be a non-subscripted fixed-point variable, and $n_1, n_2, \ldots, n_m$ must be statement numbers. If the value of the variable i at the time this statement is executed is j, then control is transferred to the statement with the statement number n_j. The value of the fixed-point variable i must be in the range of 1 to m. If the value of the fixed-point variable i is incorrectly given as zero, the object program will stop after giving an error indication; if it is greater than m, unpredicted incorrect results will occur and there will be no error stop.

For example, suppose that one of four computations should be carried out on some input data, depending on whether an input number N is 1, 2, 3, or 4. If N is 1, we wish to transfer control to a program beginning with statement 123, if it is 2 to statement 600, if it is 3 to 507, and if it is 4 to 1280. Suppose now that the data, including N, has been read in and the program is ready to transfer to the correct program. The following statement will have the desired effect:

GO TO (123, 600, 507, 1280), N

This method, although simple, has its risks in the case described, for N could be mispunched to be greater than 4. It is possible to do the same thing, but with verification of the accuracy of N, using the IF statement described below.

IF Statement — IF (e) n_1, n_2, n_3

This statement is used when a transfer of control is to be conditional, depending on whether an expression is less than zero, equal to zero, or greater than zero. In the IF statement, e is an expression and n_1, n_2, n_3 are statement numbers. Control is transferred to the statement with the statement number n_1, n_2, or n_3, depending on whether the value of the expression e is less than zero, equal to zero, or greater than zero, respectively. To illustrate one way to use the IF statement, the previous example may be reworked to include a test to be sure that N is indeed in the range of 1 to 4. Suppose that a program has been written to handle the situation of an invalid value of N and that the first statement of the error-handling program has statement number 20000. The following three statements will test the value of N for validity and transfer to the correct program if it is valid:

ALGEBRAIC COMPILER STATEMENT

TITLE _____ WRITTEN BY _____ CHECKED BY _____ DATE _____ PAGE ___ OF ___

	A, B, C	STATEMENT NUMBER	CONTINUATION	ALGEBRAIC COMPILER STATEMENT
1				IF(N) 20000, 20000, 100
2		100		IF(N-5) 101, 20000, 20000
3		101		GO TO (123, 600, 507, 1280), N

If N is less than or equal to zero, or greater than or equal to 5, control is transferred to the error program at 20000. Another way to do the same thing would involve only the use of IF statements:

ALGEBRAIC COMPILER STATEMENT

TITLE _____ WRITTEN BY _____ CHECKED BY _____ DATE _____ PAGE ___ OF ___

A,B,C	STATEMENT NUMBER	CONTINUE	ALGEBRAIC COMPILER STATEMENT
1			IF(N-1) 20000, 123, 100
2	100		IF(N-2) 20000, 600, 101
3	101		IF(N-3) 20000, 507, 102
4	102		IF(N-4) 20000, 12880, 20000

Control could never get to the second, third, or fourth IF statement if N were less than 1, so that there is really no need for the "less than zero" path on the IF statements after the first. However, there is no way to avoid writing statement numbers for all three paths even though one of them might never be followed. The choice of statement number of the error program has no significance.

Assigned GO TO Statement — GO TO n, (n_1, n_2, ..., n_m)

This statement, together with the ASSIGN statement, provides another way to effect a transfer of control, usually, at some point later in the program than when the "decision" is made. In this statement, n must be a non-subscripted fixed-point variable which appears in a previously executed ASSIGN statement, and n_1, n_2, ..., n_m must be statement numbers. Control is transferred to the statement having for its statement number whichever one of the values n_1, n_2, ..., n_m was most recently assigned to n by an ASSIGN statement.

ASSIGN Statement — ASSIGN n_i to n

In this statement, n must be a non-subscripted fixed-point variable which appears in an assigned GO TO statement, and n_i must be one of the statement numbers appearing in parentheses in the same assigned GO TO. When the assigned GO TO is next executed, control is transferred to the statement with the statement number n_i, unless another applicable ASSIGN statement intervenes.

It is important not to confuse the assigned GO TO with the computed GO TO, and, in particular, to understand that what the ASSIGN statement does for the assigned GO TO cannot be done any other way. For instance, the following two statements are not equivalent:

1. ASSIGN 208 TO N
2. N = 208 in connection with the statement:
   GO TO N, (106, 208, 200).

However, the following two programs are equivalent:

1. ASSIGN 208 TO N
   GO TO N, (106, 208, 200)
2. N = 2
   GO TO (106, 208, 200), N.

IF PARITY Statement — IF PARITY n_1, n_2

This statement may be used to alter the course of a computation upon detection of an uncorrectable parity error on a magnetic tape. In the statement, n_1 and n_2 must be statement numbers. If there was a parity error detected during the execution of the preceding input or output statement and the orthotronic routines were not able to correct it, the statement with the statement number n_1 is executed next; if there was no error or if it was corrected, the statement with the statement number n_2 is executed next. The IF PARITY statement may optionally follow any of the statements READ TAPE, WRITE TAPE, READ INPUT TAPE, or WRITE OUTPUT TAPE; if so, it must be the next executable statement. If the statements IF PARITY and IF END OF FILE are both used, the IF PARITY must be first. If an uncorrectable error is detected and there is no IF PARITY statement following the input or output statement, the object program will print an error indication and stop.

For a further discussion of this statement and its use, see Section V on input and output statements.

IF END OF FILE Statement — IF END OF FILE n_1, n_2

This statement may be used to alter the course of a computation under any of the following conditions:

1. In connection with a READ TAPE statement, upon detection of the indication written on a magnetic tape by the END FILE statement;
2. In connection with a READ, READONE, or READTWO statement, upon detection of a card with the word FINIS punched in columns 2 through 6;
3. In connection with a READ INPUT TAPE statement, upon detection of a record produced by a card with the word FINIS punched in columns 2 through 6.

In the statement, n_1 and n_2 must be statement numbers. If the relevant condition was detected in connection with the preceding input or output statement, the statement with the statement number n_1 is executed next; if the condition was not detected, the statement with the statement number n_2 is executed next. The IF END OF FILE statement must be the next executable statement after the input or output statement to which it refers, except that an IF PARITY statement may intervene. If any of the conditions listed above are detected and there is no IF END OF FILE statement following the input or output statement, the object program will produce an error indication and stop.

A further discussion of this statement and its use may be found in Section V on input and output statements.

CONTINUE Statement — CONTINUE

This statement does not generate any instructions in the object programs. It is used primarily as the last statement in the range of a DO statement (see below), when it is needed to satisfy the requirement that the range of a DO must not end with any statement which can cause a transfer of control. This statement is particularly useful since no transfer of control within the range of a DO can return to the beginning of a new cycle. Instead, control should be transferred to a CONTINUE at the end of the DO range.

DO Statement — DO n i = n_1, n_2, n_3

or DO n i = n_1, n_2

This is a very powerful statement which makes it possible to carry out repetitive procedures, often (but not always) working with the elements of arrays. After describing how the statement operates and some rules which must be observed in using it, we shall give a number of examples of the use of this important feature of the Algebraic Compiler language.

In the DO statement, n must be a statement number, i must be a non-subscripted fixed-point variable, and n_1, n_2, and n_3 must each be either an unsigned fixed-point constant or a non-subscripted fixed-point variable. If n_3 is not stated, as in the second form of the statement, it is assumed to be 1.

The statements following the DO, up to and including the statement with the statement number n, are executed repeatedly. They are executed first with i = n_1; before each succeeding execution, i is increased by n_3. Repeated execution continues until the statements have been executed with i equal to the largest value which does not exceed n_2.

The range of the DO is defined to be the set of repeatedly executed statements. In other words, it is the set of statements beginning with the first executable statement immediately following the DO statement and continuing up to and including the statement with the statement number n.

The fixed-point variable i is called the index of the DO. Throughout the execution of the range, i is available for use in computation (see Rule 3 below), either as a fixed-point variable or as a subscript. The value of i is also available for use in computation if control passes to statements outside of the range. Control may pass outside of the range of the DO either by the execution of control statements which cause a transfer of control outside or by the normal completion of the number of executions of the range as specified by the indexing parameters n_1, n_2, and n_3. In the latter case, the DO is said to be satisfied.

A few rules must be observed in writing DO statements.

Rule 1. If the range of one DO (the "outer" DO) contains statements in the range of another DO (the "inner" DO), then all statements in the range of the inner DO must also be in the range of the outer DO. This does not prohibit having the ranges of two or more DO's end with the same statement.

Rule 2. The last statement in the range of a DO must not be a statement which can cause a transfer of control. The CONTINUE statement is provided for situations which would otherwise violate this rule.

Rule 3. No statement may be executed within the range of a DO, which redefines or otherwise alters the value of the index or the indexing parameters of the DO.

Rule 4. Control must not transfer into the range of a DO from a statement outside its range. One exception to this rule is that it is permissible to transfer control out of the range of a DO, perform a series of calculations, and then transfer back to the same section of the range of the DO from which exit was made. When this is done, the statements to which control is transferred are called the extended range of the DO. It is still necessary to observe Rule 3, as though the series of calculations performed were part of the range of the DO -- which they are. If the extended range of the DO itself contains DO's, then there is a further restriction. A nest of DO's is defined to be a set of DO's with overlapping ranges; a completely nested set is one in which every pair of

DO's is such that one contains the other. With these definitions, the second part of Rule 4 states: if the extended range of a DO contains other DO's, then a transfer to the extended range is only permitted from the innermost DO of a completely nested set.

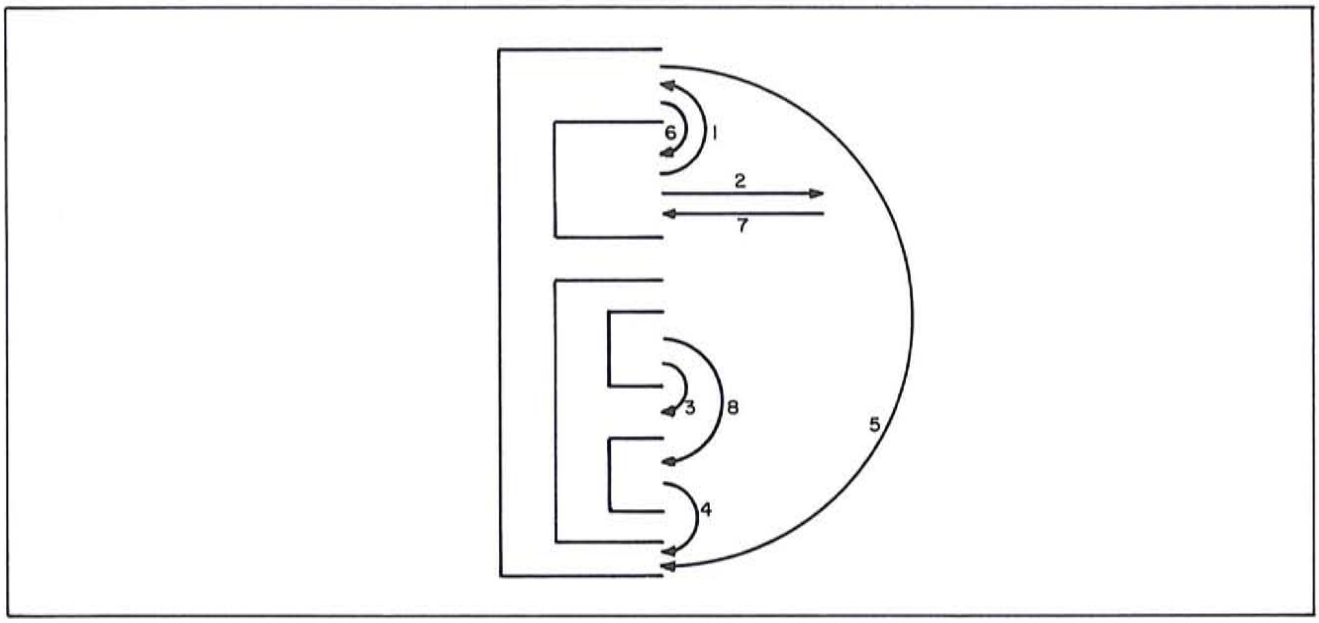


Figure 4. Transfer of Control with Respect to Sets of Non-Completely Nested DO's

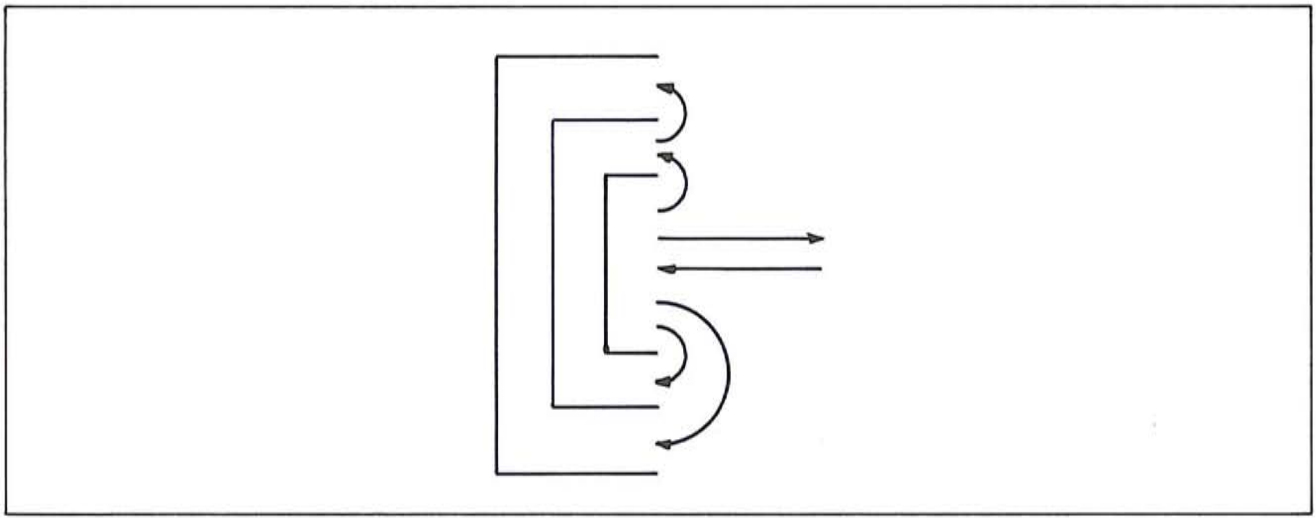


Figure 5. Transfer of Control with Respect to Completely Nested DO's

Figure 4 shows some acceptable and unacceptable transfers of control. Transfers 1 through 5 are always acceptable. Transfers 6 and 8 are always wrong. Assuming that

transfer 2 goes to a series of calculations from which transfer 7 returns, transfer 7 is correct only if the series of calculations contains no DO's, since the set of DO's is not completely nested. The DO's in Figure 5 are completely nested, and all the transfers are acceptable whether or not there are DO's in the extended range.

For a first example of the use of a DO loop, consider the scalar product that was used earlier to illustrate the use of subscripts. Recall that there were two one-dimensional arrays XVECT and YVECT, and it was required to compute the sum of the products of corresponding elements. The DO loop below accomplishes this in a manner similar to the second method used before:

ALGEBRAIC COMPILER STATEMENT

TITLE _____ WRITTEN BY _____ CHECKED BY _____ DATE _____ PAGE ___ OF ___

	A, B, C	STATEMENT NUMBER	CONTINUATION	ALGEBRAIC COMPILER STATEMENT
1				SCPROD = 0.0
2				DO 50 I = 1,3
3		50		SCPROD = SCPROD + XVECT(I) * YVECT(I)

As a matter of fact, the execution of this DO loop is almost exactly the same as the execution of the earlier example. Before, we set SCPROD = 0.0 and I = 1, and carried out SCPROD = SCPROD + XVECT(I) * YVECT(I). Then we set I = 2, and carried out the same statement, although we had to write it a second time, and then did it again with I = 3. This is exactly what the DO loop above does. With the statement DO 50 I = 1, 3 we have said, "carry out the statements down through the one with statement number 50 (in this case it is the next statement), first with I = 1, then add 1 to I (since n_3 was not stated, it is assumed to be 1) and do it again, etc., until the range has been executed with I = 3".

For another simple example of the use of the DO statement in manipulating arrays, suppose that there are two one-dimensional arrays named X and Y which are each of maximum length 100, but that the number of elements in each, for any particular set of data, is variable. The number of elements in each array is given for a particular problem by the value of the fixed-point variable M. It is required to add the two arrays, element by element, and place the elements of the sum in an array named Z. Assuming that the variable M is defined elsewhere, the following statements will carry out the required computation:

ALGEBRAIC COMPILER STATEMENT

TITLE _____ WRITTEN BY _____ CHECKED BY _____ DATE _____ PAGE ___ OF ___

A, B, C	STATEMENT NUMBER	CONTIN	ALGEBRAIC COMPILER STATEMENT
1			DIMENSION X(100), Y(100), Z(100)
2			DO 789 I = 1, M
3	789		Z(I) = X(I) + Y(I)

The DIMENSION statement specifies, for each array, that it contains at most 100 elements and, by the fact of having only one subscript, that it is a one-dimensional array. Note that it is not necessary to know how many elements there are in the arrays (as long as all three have the same number of elements), but rather the variable M is used to control the number of repetitions of the DO. M must be greater than zero, and not greater than the maximum size of the arrays. If these rules are violated, the computer's action is not predicted.

Suppose now that R and S are two-dimensional arrays of the same maximum size[1]. Let the number of rows for this problem be specified by the fixed-point constant LROWS and the number of columns for this problem be specified by LCOLS. It is required to add the two arrays, which may be thought of as matrices, element by element, and place the elements of the sum in an array named T, which naturally has the same dimensions as R and S. The techniques here are very similar to the previous problem, except that there are two DO statements.

ALGEBRAIC COMPILER STATEMENT

TITLE _____ WRITTEN BY _____ CHECKED BY _____ DATE _____ PAGE ___ OF ___

A, B, C	STATEMENT NUMBER	CONTIN	ALGEBRAIC COMPILER STATEMENT
1			DIMENSION R(10,10), S(10,10), T(10,10)
2			DO 400 I = 1, LROWS
3			DO 400 J = 1, LCOLS
4	400		T(I,J) = R(I,J) + S(I,J)

Note that the subscript controlled by the inner loop (the second DO) is varying more rapidly than the subscript controlled by the outer DO. Here, the first DO establishes that the first subscript should start at 1 and run up to the value of LROWS. With I set at 1 to start, the second DO causes the second subscript to run through the values from 1 up to the value of LCOLS. When the second DO has been satisfied, the first subscript is increased to 2, and

[1] Where the row number is given by the first subscript, and the column number is given by the second subscript.

the second DO carried out in its entirety again. In short, the second DO is carried out once for each value of the index of the first DO. Stated in matrix terms, we handle all the elements of the first row, then all the elements of the second row, etc., up to all of the elements of the last row.

In this example, it actually does not matter what order is followed in adding the elements (in most problems it does). If we were to reverse the order of the two DO's, the net result in this case would be the same, but the order of handling the elements would be different; we would handle all the elements of the first column, than all the elements of the second column, etc.

The next example shows how an IF statement can be used within the range of a DO. Suppose there is a one-dimensional array named DATA; we are required to find the largest (algebraically) of the odd-numbered elements of the array and put it in BIG. We shall put the first element of the array in BIG to begin, then successively compare it with all the other odd-numbered elements and replace the contents of BIG with any element we find which is larger than the current contents. To consider only the odd-numbered elements we make $n_3 = 2$. Suppose the array has 100 elements:

ALGEBRAIC COMPILER STATEMENT

TITLE _____ WRITTEN BY _____ CHECKED BY _____ DATE _____ PAGE ___ OF ___

	A, B, C	STATEMENT NUMBER	CONTIN	ALGEBRAIC COMPILER STATEMENT
1				DIMENSION DATA(100)
2				BIG = DATA(1)
3				DO 12 K = 3,100,2
4				IF(DATA(K)-BIG) 12,12,13
5		13		BIG = DATA(K)
6		12		CONTINUE

On each execution of the range, if the current odd-numbered element from the array is less than or equal to the current contents of BIG, we transfer down to the CONTINUE statement, but if it is larger than the current contents of BIG we replace the contents of BIG with it. The CONTINUE is required here because we do not want to execute statement 13 every time, and a transfer of control cannot return to the beginning of the range to begin a new cycle.

PAUSE Statement — PAUSE or PAUSE n

This statement is used when it is desired to interrupt the execution of statements in order to allow the computer operator to take some action. If the second form of the statement is used, n must be an unsigned fixed-point octal constant. When the statement is encountered in the object program, the following are typed out on the console typewriter:

1. The title of the main program, which appeared in the TITLE statement, which must be on the first card of every program deck;
2. The word PAUSE;
3. The octal constant n (containing as many as five digits), or nothing if the first form of the statement is used;
4. The status of the simulated sense lights and sense switches.

The machine then waits for the operator to take some action. He might remove the program from the machine, he might change the status of the simulated sense switches and continue execution of the program, or he might simply make note of a prior typewriter message and continue execution of the program without making any changes. In any case, continuing execution of the program begins with the next executable statement after the PAUSE. These actions, if done properly, will have no effect on any programs being parallel processed with this one. The optional octal constant makes it possible for the programmer to set up a message key by means of which the operator can tell what action to take based on what number is typed out. Alternatively, the programmer may type out English instructions to the operator through the use of appropriate ARGUS instructions (see Section III). The constant n is specified as octal in order to provide compatibility with systems similar to the Honeywell Algebraic Compiler.

STOP Statement — STOP or STOP n

This statement is used when it is desired to stop execution of the program, without any provision for continuation, in short, when the problem is finished or when errors in input have occurred from which there is no way to recover without starting all over again. If the second form of the statement is used, n must be an unsigned fixed-point octal constant. When the statement is encountered in the object program, the following are typed out on the console typewriter:

1. The title of the main program, which appeared in the TITLE statement, which must be on the first card of every program deck;
2. The word STOP;
3. The octal constant n (containing as many as five digits), or nothing if the first form of the statement is used;
4. The status of the simulated sense lights.

Once this information has been typed, control is automatically returned to the Executive Routine. Nothing carried out in connection with the STOP statement has any effect on any other programs being parallel processed with this one.

The octal number n can be used to give information to the computer operator, if he has been provided with a list of the possible stops and the meaning of each, including a description of action he should take for each. Alternatively, an ARGUS instruction (see Section III) may be used to type out information to the operator.

SENSE LIGHT Statement[1] SENSE LIGHT i

This statement provides a means of indicating conditions in a problem both to the operator and to other portions of the program. The value of i must lie in the range of zero through 4. If i is zero, all sense lights (1 through 4) will be turned off, i.e., SENSE LIGHT 0 in effect clears all sense lights. If i has any other value, i.e., 1 through 4, that particular sense light will be turned on. For example, SENSE LIGHT 3 turns on sense light 3. For a discussion of sense lights and sense switches, see Appendix B.

IF (SENSE LIGHT) Statement[1] IF (SENSE LIGHT i) n_1, n_2

This statement is used to alter conditionally the sequence of the execution of statements dependent upon the status of one of the sense lights. In the statement, n_1 and n_2 are statement numbers and i is the number of a sense light, 1 through 4. If sense light i is in the on condition, control is transferred to statement number n_1, otherwise control is transferred to statement number n_2. If the sense light is on at the time of execution of this statement, it will be turned off. In other words, sense light i is always left in the off condition as the result of the execution of this statement.

IF (SENSE SWITCH) Statement[2] IF (SENSE SWITCH i) n_1, n_2

This statement is similar to the IF SENSE LIGHT statement except that the sense switches (see Appendix B) are interrogated rather than the sense lights. n_1 and n_2 are statement numbers and i identifies the sense switch used. The value of i may range from 1 through 6. Control is transferred to statement n_1 if sense switch i is down and to statement n_2 if sense switch i is up.

IF ACCUMULATOR OVERFLOW Statement IF ACCUMULATOR OVERFLOW n_1, n_2

This statement is used to control the program sequence depending on the setting of a switch which is set by an addition or subtraction overflow unprogrammed transfer. Control is transferred to statement number n_1 if an accumulator overflow has occurred or to statement number n_2 if overflow has not occurred since the previous IF ACCUMULATOR OVERFLOW statement. The use of this statement resets the internal indicator tested.

IF QUOTIENT OVERFLOW Statement IF QUOTIENT OVERFLOW n_1, n_2

This statement is used to test the status of a switch set by an exponential overflow or underflow unprogrammed transfer. Control is transferred to statement n_1 if an exponent has

[1] SENSE BIT and IF (SENSE BIT) Statements may be used interchangeably with the SENSE LIGHT and IF (SENSE LIGHT) Statements respectively.

[2] The IF (SENSE FLAG) Statement may be used interchangeably with the IF (SENSE SWITCH) Statement.

been created by any of the floating-point operations that is greater than +63 or less than -64 or to statement n_2 if these exponent limits have not been exceeded since the previous IF QUOTIENT OVERFLOW statement. The use of this statement resets the internal indicator tested.

IF DIVIDE CHECK Statement — IF DIVIDE CHECK n_1, n_2

This statement is used to test a switch set by a division overcapacity unprogrammed transfer. Control is transferred to statement n_1 if a division instruction has been attempted that cannot be performed or to statement n_2 if no illegal divisions have been attempted since the previous IF DIVIDE CHECK statement. The use of this statement resets the internal indicator tested.

TITLE Statement — TITLE Name

This statement is used to provide each program with a name by which it may be referred to in connection with the collector tape, discussed in Section VI. This statement must be on the first card of every program. The word TITLE must be punched in columns 2 through 6 of the statement card, and the desired title in columns 7 through 14. The name used should not duplicate any already on the collector tape. If no TITLE statement is provided, a dummy name will be supplied by the Compiler; it will ordinarily be desirable to correct the omission, and the dummy name makes it at least possible to make reference to the compiled program in order to change the name. A TITLE statement is not required for FUNCTION and SUBROUTINE subprograms (see Section VI), and in fact is irrelevant; the name of the subprogram becomes its name on the collector and any TITLE card is ignored. A TITLE card must not be preceded by a blank card.

END Statement — END

This is a non-executable statement which must appear at the end of every program or subprogram deck. It is required in order to separate programs in batch compilation, but it is nevertheless required in every program, even if the "batch" consists of only one program. The word END must be punched in columns 7 through 9 of the statement card; anything else on the card is ignored. An END card must not be followed by a blank card.

FINIS Statement — FINIS

This is a non-executable statement which must appear at the end of the deck of programs being batch compiled. It is required even if the batch consists of only one program. The word FINIS must be punched in columns 2 through 6 of the card, and the remainder of the card must be blank. It may help to note that a FINIS statement is always preceded by an END statement, although an END statement is not always followed by a FINIS statement; if several programs are being batch compiled, a FINIS appears only at the end of the entire batch.

SECTION V

INPUT AND OUTPUT STATEMENTS

There are a number of considerations which must be included in planning and writing input and output statements. Some or all of the following factors are involved in these statements:

1. The choice of input or output device. This is handled by choosing the appropriate statement from: READ; READONE; READTWO; PRINT; PRINTONE; PRINTTWO; PUNCH; PUNCHONE; PUNCHTWO; READ TAPE; READ INPUT TAPE; WRITE TAPE; and WRITE OUTPUT TAPE.

   Four other statements are classified as input-output statements on the basis that they involve input-output devices or have a form similar to input-output statements but they do not actually transmit any information. These are: ERASE; END FILE; REWIND; and BACKSPACE.

2. The determination of what information is to be transmitted. This is handled by the use of a list. This list specifies the names of the variables to be read in or written out. (The word "list" is used here in a specialized sense which is discussed at length below.)

3. The arrangement of the information on the input medium (cards, magnetic tape), or the arrangement on the output medium (cards, printer, tape). This is handled by providing suitable information in a FORMAT statement. A number of the input-output statements are not required to reference a FORMAT statement, either because they do not transmit information or because the format of the information is fixed.

4. The type of conversion to be applied between the external and internal information. This is handled by writing an appropriate field specification in a FORMAT statement.

In this section we shall first discuss what is meant by a list, then the FORMAT statement and how it operates in conjunction with a list, then the field specifications (which covers both format control and conversion type), and finally each of the input and output statements will be discussed in turn.

Definition of a List

Any input or output statement which actually transmits variables requires a list, in order to specify the variables to be transferred between storage and the input or output device, and to specify the sequence in which they are to be transferred. The simplest type of list consists simply of the names of the variables to be transferred. For instance, if to read a card we write:

```
READ 200, A, B, C(1), C(2), K78
```

where 200 would be the statement number of a FORMAT statement, then the list consists of the names of the variables A, B, C(1), C(2), K78. The first five fields on the card, as defined in the FORMAT statement, would be read into the storage locations assigned to the variables named, with the first field being taken as A, the second as B, etc. As we shall see, the FORMAT statement, in addition to specifying the length of each field, would in effect state how to expect to find the numbers punched in the fields.

It is permissible to use fixed-point variables which appear in a list as subscripts elsewhere in the same list. One way in which this flexibility might be useful would be in reading elements of an array which appear in a deck of cards in random order. Suppose, for instance, that the elements of a two-dimensional array (matrix) are punched one to a card, with the row and column number of each element punched on the same card with it, in the order I, J, DATA(I, J). The list is I, J, DATA(I, J). Assuming that 300 refers to a FORMAT statement which specifies that the first two numbers on the card are to be taken as fixed point and the third as floating point, the READ statement is:

```
READ 300, I, J, DATA(I, J).
```

When this is done with input, however, the variables used as subscripts must appear in the list as input variables before they appear as subscripts. We shall see immediately below that there is another way in which fixed-point variables can be used in a list which is somewhat similar to this technique.

When parts of arrays or entire arrays are to be transferred, it is often not necessary to name each element explicitly. To transfer an entire array, it is only necessary to name the array in a list without any subscripts. The name of the array must, of course, appear in a DIMENSION statement, but in the list it need not carry any subscripting information.

When only certain elements of an array are to be transferred, it is often possible to specify them in the list by giving indexing information in a way which parallels a DO loop, although it is not literally a DO loop. This is done by enclosing the indexed variables in parentheses and giving the indexing information just before the closing parenthesis. For instance, the statement:

```
READ 400, (DATA(1, I), I = 1, 10)
```

would call for 10 numbers to be read from cards and stored as the first 10 elements in the first row of the matrix DATA. The same 10 elements could be read in as main diagonal elements by the statement:

```
READ 400, (DATA(I, I), I = 1, 10).
```

Just as it is possible to have nests of DO's, it is possible to have nests of up to a maximum of three indexed variables in a list. Suppose, for an example, that we wanted to read 50

numbers from cards, taking the first 10 as the first 10 elements of the first row of DATA, the second 10 as the first 10 elements of the third row of DATA, the next 10 as the first 10 elements of the fifth row of DATA, etc. It could be done with the statement:

```
READ 400, ((DATA(I, J), J = 1, 10), I = 1, 9, 2).
```

In a certain sense, this statement may be thought of as equivalent to the DO loop:

```
    DO 10 I = 1, 9, 2
    DO 10 J = 1, 10
10  DATA(I, J)
```

In this example, statement 10 is to be understood in the sense: "The next number read is to be taken as DATA(I, J)." Thus it is seen that the idea of "inner" and "outer" DO's corresponds to the inner and outer indexing information in the list.

It has been noted earlier that fixed-point variables which appear in the list may be used elsewhere in the list as indices; on input, the appearance must be earlier in the list than their use as indices.

To illustrate some of these ideas, suppose that the following list is specified with an input statement, and that the value of the integer N which is read in is 2:

```
A, N, B(N), (C(I), D(I, 2), I = 1, N),
((E(I, J), I = 1, 2), F(J), J = 1, 5, 2).
```

The variables read in would be in the following sequence:

```
A, N, B(2), C(1), D(1, 2), C(2), D(2, 2),
E(1, 1), E(2, 1), F(1), E(1, 3), E(2, 3), F(3),
E(1, 5), E(2, 5), F(5).
```

The effect of the list would be that of the following implicit DO loop, again assuming that the value of N read in is 2:

```
 1.  A
 2.  N ( = 2 by assumption)
 3.  B(2)
 4.  DO 6 I = 1, N
 5.  C(I)
 6.  D(I, 2)
 7.  DO 10 J = 1, 5, 2
 8.  DO 9 I = 1, 2
 9.  E(I, J)
10.  F(J)
```

FORMAT Statement FORMAT (Field Specifications)

All of the input and output statements which require a list, with the exception of READ TAPE and WRITE TAPE, require, in addition, the statement number of a FORMAT statement which describes the information format to be used. The FORMAT statement also describes, in some cases, the kind of conversion to be performed between the internal and external representation of the information to be transferred. A FORMAT statement is not executable, i. e., does not by itself cause any action in the object program, and may be placed anywhere in the source program.

In the discussions which follow, the term unit record is used for generality. Depending on which input or output statement is used, a unit record may consist of:

1. A line to be printed on an on-line printer, with a maximum of 120 characters;
2. A punched card to be read from an on-line card reader or punched on a directly-connected punch, with up to 80 characters;
3. An alphabetic tape record to be read or written, with a maximum of 120 characters;
4. A binary type record to be read or written, with any number of words; a unit record may be any number of physical records on tape; this is handled automatically.

The field specification in a FORMAT statement describes the unit record(s) involved by giving, for each field in the record, beginning with the first character of the record:

1. The type of information and/or the type of conversion to be used; this is done with the seven field specification characters discussed in detail below;
2. The number of characters in the field;
3. For some of the field specifications certain other information is required or may optionally be given; these cases are discussed below, in connection with the field specifications.

To give some short examples before proceeding to the details:

1. If the statement FORMAT(F10.4) were used in connection with input, the "F10.4" would mean that a 10-column field consists of a decimal number punched without an exponent, with four places after an understood decimal point, and is to be converted to a floating-point variable;
2. If the statement FORMAT(I5) were used in connection with output, the "I5" would mean that a fixed-point (i. e., integer) variable is to be written out into a five-character field in the external medium, with the integer placed in the right side of the field, and with a minus sign immediately to the left of the number if it is negative.

If a number of consecutive fields are to be treated under the same field specification, it is permissible to write the number of such fields before the field specification characters.

Thus, "5F10.4" would mean that there are five fields of the type described in the first example above. It is also possible to call for the repetition of groups of fields, by enclosing the group of field specifications within parentheses and writing the desired number of repetitions in front of the opening parenthesis. For example, suppose that a unit record consists of fields described by I3 and F10.4 alternately, with eight such pairs. The easy way to define such a situation would be 8(I3, F10.4). Note that this is not equivalent to 8I3, 8F10.4; the latter would mean eight consecutive I3 fields, then eight consecutive F10.4 fields, instead of the intended alternation of the two types. Only one level of such grouping is permitted, i. e., parentheses within parentheses are not permitted for this purpose.

When the list of an input or output statement is used to transfer more than one unit record, with the different records having different formats, a slash (/) is used to separate the format specification of each record. If for example the statement:

FORMAT (10I3/8F10.6)

were used with a READ statement, the effect would be to read one card under control of the 10I3, and a second under control of 8F10.6. It is possible to specify that the first one or more records have a special format, and that all following records have the same format; this is done by enclosing the last record specification in a second set of parentheses. A slash always indicates the end of one record and beginning of a new one; the closing parenthesis of the FORMAT statement always indicates the end of a record. The skipping of entire records, which in practice usually means the skipping of lines on a printer, is called for by writing successive slashes. The skipping of n records is called for by writing n + 1 slashes.

With the exceptions of a FORMAT statement which consists entirely of Hollerith fields and of the case of the "blank" field specification (see below), a FORMAT statement is always used in conjunction with the list of an input or output statement. The list specifies the variables to be transferred and in what sequence, and the associated FORMAT statement specifies the format of each variable as well as the length of each record, if there is more than one. As the object program transmits the variables named in the list, it scans the FORMAT statement, from left to right, to find the proper field specification for each variable, taking into account any repetition of field specifications or groups of field specifications. Whenever Hollerith field specifications are encountered in scanning the FORMAT statement, they are dealt with in the proper place, without any transmission of variables from the list. The transmission of variables is terminated only when all items in the list have been transmitted, but any remaining Hollerith fields will be dealt with even after the transmission of the last variable specified in the list. If the last field specification in the FORMAT statement has

been used and items named in the list remain to be transmitted, the closing parenthesis indicates the end of unit record, and scanning of the FORMAT statement begins again with the first field specification after the last open parenthesis in the statement.

To illustrate a simple case of scanning the FORMAT statement to find the field specification corresponding to each variable, consider the following:

ALGEBRAIC COMPILER STATEMENT

TITLE ____ WRITTEN BY ____ CHECKED BY ____ DATE ____ PAGE ____ OF ____

A, B, C	STATEMENT NUMBER	CONTIN	ALGEBRAIC COMPILER STATEMENT (1 6 11 23 38 52 66 72 80)
1			PRINT 25, A, B, C
2	25		FORMAT(F12.4, F13.4, F14.4)

The variable A would be associated with F12.4, B with F13.4, and C with F14.4. On the other hand, the following situation is also called scanning:

ALGEBRAIC COMPILER STATEMENT

TITLE ____ WRITTEN BY ____ CHECKED BY ____ DATE ____ PAGE ____ OF ____

A, B, C	STATEMENT NUMBER	CONTIN	ALGEBRAIC COMPILER STATEMENT (1 6 11 23 38 52 66 72 80)
1			PRINT 125, A, B, C
2	125		FORMAT(3F16.6)

In other words, if in the scanning of the FORMAT statement a repeated field is found, it is used as many times as the repetition number. In the above case, the printed output would appear on one line. If the repetition number were not included, three separate lines would occur, and the FORMAT statement would appear as:

ALGEBRAIC COMPILER STATEMENT

TITLE ____ WRITTEN BY ____ CHECKED BY ____ DATE ____ PAGE ____ OF ____

A, B, C	STATEMENT NUMBER	CONTIN	ALGEBRAIC COMPILER STATEMENT (1 6 11 23 38 52 66 72 80)
1			PRINT 225, A, B, C
2	225		FORMAT(F16.6)

In this case there remain items to be transmitted when the last field specification has been used, and scanning will begin again with the first (and in this case only) field specification after the last open parenthesis in the statement. Each number will appear on a separate line since the closing parenthesis indicates the end of unit record. The F16.6 is used for all variables transmitted, no matter how many there are.

Here is an example of a case where the repetition number is required:

ALGEBRAIC COMPILER STATEMENT

TITLE ☐☐☐☐☐☐☐☐ WRITTEN BY _____ CHECKED BY _____ DATE _____ PAGE ___ OF ___

A, B, C	STATEMENT NUMBER	CONTINUATION	ALGEBRAIC COMPILER STATEMENT
			1 6 11 23 38 52 66 72 80
1			PRINT 325, L, A, (DATA(I), I=1,4), M
2	325		FORMAT (I5, F10.2, 4F15.5, I7)

Here, I5 is associated with L, F10.2 with A, F15.5 with the four values of DATA(I), and I7 with M. Since the four field specifications are different, there would be no way to make use of repeated scanning.

The following is a case where repeated scanning can be used. Suppose that we are reading a deck of cards which consists mostly of elements of a one-dimensional array punched one to a card, but where the first card contains a fixed-point number N which specifies how many elements there are. Suppose the first card has N punched in the first three columns, and that on the element cards the element number is punched in the first three columns and the element in the next 12 columns. The following two statements would call for the entire deck to be read and each element stored in the proper place:

ALGEBRAIC COMPILER STATEMENT

TITLE ☐☐☐☐☐☐☐☐ WRITTEN BY _____ CHECKED BY _____ DATE _____ PAGE ___ OF ___

A, B, C	STATEMENT NUMBER	CONTINUATION	ALGEBRAIC COMPILER STATEMENT
			1 6 11 23 38 52 66 72 80
1			READ 425, N, (J, ARRAY(J), I=1,N)
2	425		FORMAT(I3/(I3, F12.6))

This illustrates a number of points. The READ statement first gets N, which is associated with the first I3; the slash in the FORMAT statement indicates that after N there is nothing more on that card. In the READ statement the parentheses indicate variables with indexing information supplied. On each card after the first, the READ statement expects to find a fixed-point number and a floating-point number, and expects to find N such cards because of the indexing information. The indexing information here is a little different from what we have had before; here, J is read from the card and then used immediately to determine where in the array to store the floating-point number. The indexing parameter I is used only to control the total number of times the process should be repeated, and is not employed as a subscript. In the FORMAT statement, when the last field specification (F12.6) has been used, scanning begins again with the second I3, not the first, because of the rule about

parentheses in the FORMAT statement. This is how we want it, because otherwise we would always be expecting to find two fixed-point and then one floating-point number, which is the case only at the beginning of the deck. To review: the slash in the FORMAT statement is necessary to indicate that the first record ends after the one number; a slash is not necessary after the F12.6, because the closing parenthesis of the FORMAT statement always indicates the end of a record.

Another example of FORMAT statement scanning appears after the discussion of the Hollerith field specification.

Scale Factor

Before investigating the various field specification types in detail, we must mention a matter which applies to two of them. That is the optional use of a scale factor with the "E" and "F" field specifications. This is done by writing "sP" before the field specification, where s is the scale factor. Examples of their use appear below; here we state only a few general considerations in order to avoid repeating them with the two discussions to which they apply.

1. Once a scale factor has been given, it applies to all "E" and "F" field specifications in the same FORMAT statement, until another scale factor appears in the scanning of the statement;
2. If no scale factor is given, it is taken to be zero. Once a non-zero scale factor has been given, a scale factor of zero must be given in order to return to the "normal" mode;
3. Scale factors apply only to the "E" and "F" field specifications, and with the "E" type only to output. Use of the scale factor with any other field specification or with input on the "E" type has no effect;
4. When a scale factor is written with a field specification which includes a repetition number, the repetition number is written between the scale factor and the E or the F. If there is no repetition number, i. e., if it is understood to be 1, then it may be written or not. Thus, with the "F" field specification, for instance, the following are all permissible:

   3P4F12.4, 3PF12.4, 3P1F12.4,

   the last two are equivalent.

Field Specification "E" (Floating Point) Ew.d

The "E" field specification is used to indicate conversion between an internal floating-point number and an external floating-point number, i. e., one written with an explicit exponent. The total number of characters in the field in the external medium, including sign, decimal point, exponent, and any blanks, is specified by w. The number of decimal places after the decimal point (not counting the exponent) is specified by d; d is treated modulo 10, i. e., only the last digit is used if more than one digit is written. The field

specification applies both to input and output, of course, but since the usage is somewhat different between input and output, we shall describe them separately.

Input Data Preparation

A sign, if it appears, must be the first non-blank character of the field. The use of a + is optional; i.e., if no sign is punched, the number is taken to be positive. The use of a decimal point is optional; if it is not supplied, then the position of the assumed decimal point, counted from the right, is given by d, but if a decimal point is supplied, then its position overrides d. Blanks embedded in the number are taken to be zeros. The "number" part of the field must not exceed 12 digits, not counting sign or decimal point. The exponent part of the field is of the general form E±ee, where ee is the numeric exponent, but several simplifications for convenience in punching cards are permitted.

A positive exponent may appear with the + omitted or replaced with a blank, i. e., in the forms E ee or Eee. If the first digit of the exponent is zero, it may be omitted. If the exponent appears with a sign, the E may be omitted. Thus, the following are all permissible (and equivalent) forms for the exponent plus 2:

E + 02, E 02, E02, E + 2, E2, +02, +2.

A scale factor has no effect on input with the "E" field specification.

For a first illustration, observe that the following numbers all convert to the same floating-point number if read in under the control of E14.0 (remember that a decimal point in the field overrides d in the field specification):

+1234.5678E04, 1.2345678+7, 12345678.E0, 123456780.-1

With the same reminder, the following numbers all convert to the same floating-point number under control of E14.7:

-12345678+0, -1.2345678+0, -1234.5678E-03, -0.12345678+01

Output Data Presentation

The number will appear at the right of the field if w is larger than the number of characters in the field. If w is not large enough to contain the converted internal number, leading characters will be lost and no indication of the fact given. There will be no embedded blanks in the field, with the

exception that + signs are not entered but are replaced with blanks. In the absence of a scale factor, the field will appear in the form ±0.nn...E±ee (except that any + signs do not appear), where the number of places after the decimal point is specified by d.

A positive scale factor may be used, by writing the field specification in the form sPnEw.d, where s is the scale factor, and n is the repetition number. The effect of the use of the scale factor in this case is to move the decimal point s places to the right and to decrease the exponent by s. (The effect of a scale factor when used with the "F" field specification is different). Recall also that once a scale factor is given it continues to apply to all succeeding "E" and "F" field specifications in the same FORMAT statement until another scale factor appears.

To illustrate, suppose that we have in storage three numbers which if printed under control of 3E17.8 would appear as:

```
 0.12345678E 03  -0.44444444E 00   0.87654321E-04
```

The same numbers printed under control of 3E13.4 would appear as:

```
 0.1234E 03  -0.4444E 00   0.8765E-04
```

The same numbers printed under control of 1P3E9.2 would appear as:

```
1.23E 02-4.44E-01 8.76E-05
```

Note that by allowing only the minimum number of places in the field, we have crowded the numbers together. The numbers are not rounded. The same numbers printed under control of 7P3E20.1 would appear as:

```
      1234567.8E-04       -4444444.4E-07       8765432.1E-11
```

Field Specification "F" (External Fixed Point) Fw.d

The "F" field specification is used to indicate conversion between an internal floating-point number and an external fixed-point number, i, e., one written without an exponent. The total number of characters in the field, including sign, decimal point, and any blanks, is specified by w. The number of decimal places after the decimal point is specified by d; d is treated modulo 10, i. e., only the right-hand digit is used if more than one digit is written.

Input Data Preparation

A sign, if it appears, must be the first non-blank character of the field. The use of a + sign is optional; a number written without a sign is taken to be

positive. The use of a decimal point is optional; if it is not supplied, then the position of the assumed decimal point, in terms of the number of digits to its right, is given by d, but if it is supplied its position overrides d. Blanks embedded in the number are taken to be zeros. The number must not exceed 12 digits, not counting sign, blanks, or decimal point. Shown below are some sample numbers, and in parentheses the number to which they would convert if read in under control of F10.4:

```
+12345678        (+1234.5678)
 1234.5678       (+1234.5678)
-1.2345678       (-1.2345678)
.012345678      (+.012345678)
    -1.2             (-1.2)
  +123456         (+12.3456)
```

A scale factor may be used with the "F" field specification for input. The effect of a scale factor in this case is to multiply the external number by 10 to the negative of the scale factor:

$$\text{Internal number} = \text{External number} \cdot 10^{-s}$$

A scale factor in this case may be positive or negative. To illustrate, the specification 2PF10.4 would convert some of the numbers displayed above, as shown in parentheses:

```
+12345678        (+12.345678)
  +123456         (+0.123456)
```

With the specification -2PF10.4:

```
+12345678        (+123456.78)
  +123456          (+1234.56)
```

Output Data Presentation

The number will appear at the right of the field, if w is larger than the number of characters in the field. If w is not large enough to contain the converted internal number, characters at the left will be lost and no indication given. Positive numbers appear without a + sign. A positive or negative scale factor may be used by writing the field specification in the form ±sPnFw.d, where a + sign is optional, s is the scale factor, and n is the number of repetitions of the field specification. The effect of the use of the scale factor in this case is to move the decimal point of the external number s places to the right if s is positive, or the left if s is negative. As a formula, s is a number such that:

$$\text{External number} = \text{Internal number} \cdot 10^{s}$$

For examples, consider the numbers 3.14159265, 2.7182818, and -39.478418 all assumed to be in storage as floating-point variables. With the field specification 3F15.5, they would appear on output as:

```
   3.14159        2.71828       -39.47841
```

With the field specification 3F8.2, they would appear as:

```
 3.14    2.71  -39.47
```

If we used 3F8.6, we would be in trouble, because there would not be enough room for the third number. What would appear would be:

```
3.1415922.7182819.478418
```

This illustrates that in using the "F" field specification it is essential to know the maximum size of the numbers which will be written out, and allow enough space for them. With the field specification 4P3F10.0, the numbers would appear as:

```
 31415.    27182.   -394784.
```

With the field specification -2P3F6.1, the numbers would appear as:

```
 .0    .0   -.3
```

This illustrates that by making the number of decimal places too small, all significant figures can be lost.

Field Specification "I" (Integer) Iw

The "I" field specification is used to indicate conversion between an internal fixed-point integer and an external decimal integer. The total number of characters in the field, including sign and any blanks, is w.

Input Data Preparation

A sign, if it appears, must be the first non-blank character in the field. The use of a + sign is optional; if no sign appears, the integer is taken to be positive. The use of a decimal point is, of course, not permitted. Embedded blanks are taken to be zeros. The number, not counting sign, must not exceed 2^{44}, which is approximately 10^{13}. For many purposes, the practical limit is 2^{15}, i. e., for purposes of subscripting or indexing in a DO loop.

Output Data Presentation

The integer will appear at the right of the field if w is larger than the number of characters in the field. If w is not large enough to contain the converted internal number, the sign and high-order digits will be lost and no indication given. Positive integers appear without the + sign.

Field Specification "H" (Hollerith) wH

The w characters immediately following the letter H, where w may be any integer not exceeding the size of the unit record, are placed in the record in the position indicated by the position of the Hollerith field specification in the FORMAT statement. The Hollerith field specification does not call for the output of any variables, but the output of the following text itself. Any Honeywell 800 character may be used, including the character blank; this is the only instance in which a blank in a statement is not simply ignored. Indication of the presence of Hollerith text is not required in the list of the output statement which refers to the FORMAT statement containing the Hollerith field specification. Whenever a Hollerith field specification is encountered in the scanning of the FORMAT statement, the following text is written out and scanning continues without any variable having been transmitted. The Hollerith text is not available to the programmer for use in any other way than for input and output.

The characters printed by the high-speed and standard-speed printers available for the Honeywell 800 are different in a few cases. Reference should be made to the character-configuration table in Appendix A to determine what the differences are and what characters are printed by the two printers.

It is possible to write a record consisting entirely of Hollerith text by putting nothing but Hollerith text in a FORMAT statement and by giving no list with the output statement.

For all output statements that result in printing, e. g., PRINT and WRITE OUTPUT TAPE, single spacing of the printed lines will result unless specific control is given otherwise. This is accomplished through the use of the Hollerith field specification in a FORMAT statement. If a field specification 1H is used as the first field specification in a FORMAT statement associated with an output statement, no data per se is transmitted, but rather the one Hollerith character is interpreted as a control character. The permissible characters and their interpretation are:

Blank - single space after the current line is printed
+ - suppress spacing after the current line is printed
0 - double space after the current line is printed
1 - space to head of form after the current line is printed
2-9 - this number of lines are to be spaced after the current line is printed

If any other character is used in this connection, it will be placed in the output area. If this Hollerith field specification is used in connection with punching, this control character will not be punched.

In every case, the spacing information applies only to the spacing between the current line and the next one and does not carry over to any subsequent lines.

To illustrate this specialized usage, consider the following:

ALGEBRAIC COMPILER STATEMENT

TITLE ☐☐☐☐☐☐☐☐ WRITTEN BY _____ CHECKED BY _____ DATE _____ PAGE ___ OF ___

	A, B, C	STATEMENT NUMBER	CONTIN	ALGEBRAIC COMPILER STATEMENT (1 / 6 / 11 / 23 / 38 / 52 / 66 / 72 / 80)
1				PRINT 56, A, B, C
2		56		FORMAT (1H1, 3F16.6)

The effect of this statement pair is to print the line containing the three fields A, B and C and then space to the head of the next form or page.

When a FORMAT statement containing Hollerith text is referenced by an input statement, the listed text is replaced by whatever text appears in the corresponding field of the input record. If the same FORMAT statement is later used with an output statement, the text which has been "read into" the FORMAT statement will then be transferred to the output record. The text thus originated is still not available to the programmer for use in any other way than for input and output. (The "A" field specification described below is available for use in entering alphabetic data which can then be manipulated by the program.)

To illustrate the scanning of FORMAT statements containing Hollerith fields, consider the following:

ALGEBRAIC COMPILER STATEMENT

TITLE ☐☐☐☐☐☐☐☐ WRITTEN BY _____ CHECKED BY _____ DATE _____ PAGE ___ OF ___

	A, B, C	STATEMENT NUMBER	CONTIN	ALGEBRAIC COMPILER STATEMENT (1 / 6 / 11 / 23 / 38 / 52 / 66 / 72 / 80)
1				PRINT 600
2		600		FORMAT(33H WING FLUTTER CALCULATION TYPE 3///)
3				PRINT 601, ICASE, (X(J), Y(J), Z(J), J=1,3)
4		601		FORMAT(4HCASE I3/(4H X= F12.3, 4H Y= F12.3, 4H Z= F12.3))

The first FORMAT statement and PRINT statement pair will print a page heading, and space three lines, two of which occur because of the three slashes, and the third, because of the closing parenthesis. The second pair calls for an identifying number to be printed on the first body line, with the data going on following body lines with identifying information to be printed before each number. The first lines printed on a page by this

short program might look like:

```
  WING FLUTTER CALCULATION TYPE 3

CASE 24
  X=     5630.818  Y=      -78.903  Z=      889.654
  X=     6793.073  Y=      -79.005  Z=     1009.462
  X=     5571.794  Y=      -77.238  Z=      850.670
```

Notice that "CASE 24" prints on a separate line because of the slash in the second FORMAT statement. The blanks in the text following the 4H field specifications are deliberate, and results in the two spaces which separate the text from the previous number. Note that the parentheses in the second FORMAT statement cause the field specifications enclosed in the parentheses to be scanned repeatedly, until all the variables in the list have been transmitted.

Field Specification "O" (Octal) Ow

The "O" field specification is used to indicate conversion between an internal 48-bit Honeywell 800 word and an external fixed-point octal integer. The total number of characters in the field, including sign and any blanks, is w, which must not exceed 16.

Input Data Preparation

If the field consists of 16 digits, it must not have a sign; it is then converted to the 48-bit representation of the number and stored as such, i. e., with the four bit positions of the sign of the word not handled any differently from the other 44 bit positions of the word. If the field has fewer than 16 characters and appears without a sign, the conversion is handled as though enough zeros were appended at the right of the field to make a total of 16 digits. If the field has fewer than 16 characters and appears with a - sign, then the converted number is placed at the right-hand end of the word and all four sign bits are set to zero; if the field consists of a - sign and 15 digits, the leading digit must not exceed 3. If the field has fewer than 16 characters and appears with a + sign, then the converted number is placed at the right-hand end of the word and all four sign bits are set to 1; if the field consists of a + sign and 15 digits, the leading digit must not exceed 3. If an 8 or 9 or any other illegal character appears, it will be converted to its alphanumeric representation, the low-order three bits stored for that character, and no error indication given.

Output Data Presentation

If w is 16 or less, the right w octal digits of the converted 48-bit number will be written without sign. If w is greater than 16, the sign bits will be treated as sign bits, the number being written with a plus sign if any one or more of the sign bits is 1, and with a minus sign if the sign bits are all zero; the number will be written into the right side of the field.

For an example, suppose that there is in storage a number which in binary would appear as:

110 100 000 001 010 011 100 101 110 111 101 010 111 000 110 101

The four leftmost bits (binary digits) are called the sign bits of the word; in our case they are sometimes treated as sign bits and sometimes not. The following list shows several "O" formats and how they would cause the number above to be written out:

O16	6401234567527065
O17	+001234567527065
O10	4567527065
O3	065

Suppose that there is another number which in binary is:

000 011 111 110 101 100 011 010 001 000 011 101 100 010 010 010

The same field specifications would produce:

O16	0376543210354222
O17	-376543210354222
O10	3210354222
O3	222

Field Specification "A" (Alphabetic) Aw

The "A" field specification is used to indicate conversion between an internal 48-bit Honeywell 800 word, considered as the alphanumeric representation of eight Honeywell 800 characters, and an external field consisting of any combination of eight or fewer Honeywell 800 characters

Input Data Preparation

If w is less than 8, the field will be stored in left-justified form, i. e., the first character of the field will appear in the leftmost character position of the computer word, and the extra characters at the right end of the computer word will be filled with blanks.

Output Data Presentation

If w is less than 8, the w characters at the left end of the computer word will be written.

To illustrate one possible usage of this field specification, suppose that 12 five-character identification words have been punched on a card, and are to be read in as the 12 elements of a one-dimensional array named TYPE. It is desired later, on output, to print one of these identifying words with each element of another array named DATA; if the first element of DATA is printed, then the first element of TYPE is to be printed following it, etc. The five-character identifying word is to be printed in parentheses. The 12 five-character groups could be read in with:

ALGEBRAIC COMPILER STATEMENT

TITLE ☐☐☐☐☐☐☐☐ WRITTEN BY _____ CHECKED BY _____ DATE _____ PAGE ___ OF ___

A, B, C	STATEMENT NUMBER	CONTIN.	ALGEBRAIC COMPILER STATEMENT (1 6 11 23 38 52 66 72 80)
1			READ 700, TYPE
2	700		FORMAT(12A5)

The READ statement could have been READ 700, (TYPE(I), I = 1, 12), but this is not necessary; recall that an entire array may be moved simply by giving its name without subscripting information. Now, to print out an element of DATA and the corresponding element of TYPE in parentheses, we could write:

ALGEBRAIC COMPILER STATEMENT

TITLE ☐☐☐☐☐☐☐☐ WRITTEN BY _____ CHECKED BY _____ DATE _____ PAGE ___ OF ___

A, B, C	STATEMENT NUMBER	CONTIN.	ALGEBRAIC COMPILER STATEMENT (1 6 11 23 38 52 66 72 80)
1			PRINT 701, DATA(I), TYPE(I)
2	701		FORMAT (E20.8, 3H (/A5, 1H))

This must be read very carefully. In the FORMAT statement, the two blanks after the 3H are deliberate, being intended to separate the left parenthesis from the number. The left parenthesis then is text, not a controlling parenthesis in the FORMAT statement. The same comment applies to the first right parenthesis after the 1H which could easily be misread. A typical line

printed by these statements might be:

-0.80402197E-04 (GAS08)

Field Specification "B" (Blank) [1] wB

Input Data Preparation

On input, the "B" field specification calls for the next w character positions in the input record to be skipped over. No indication is required in the list of the input statement referencing the FORMAT statement.

Output Data Presentation

On output, the "B" field specification calls for w blanks to be inserted into the output record. No indication is required in the list of the output statement referencing the FORMAT statement.

This field specification does nothing that cannot be done in other ways, but it is often a considerable convenience. One common use is in avoiding long Hollerith field specifications to introduce long strings of blanks.

READ Statement READ n, List

In this statement, n is the statement number of a FORMAT statement, and the list is as described previously. The READ statement calls for the reading of cards from the on-line card reader designated as number 1. As many cards are read as are required to supply the amount of information specified in the list and the FORMAT statement. The arrangement of information on the cards is defined in the FORMAT statement; each field is converted, also as defined in the FORMAT statement, and placed in the computer storage location assigned to the corresponding variable named in the list.

If, when the READ statement is executed, the card in the card reader has the word FINIS punched in columns 2 through 6, the program will expect to find an IF END OF FILE statement immediately following the READ. This provides a simple way to signal the end of a deck which consists of a variable number of cards, because the IF END OF FILE statement allows one to alter the flow of control upon detection of the end-of-file condition. If this condition arises and there is no IF END OF FILE statement immediately following the

[1] The field specification "X" may be used interchangeably with field specification "B".

READ, the object program will give an error indication and stop. It is permissible to have more cards following the one with FINIS in columns 2 through 6, so that the FINIS card may be used to separate groups of data cards into files.

READ ONE Statement — READ ONE n, List

This statement is exactly equivalent to the READ statement.

READ TWO Statement — READ TWO n, List

This statement is equivalent to the READ and READ ONE statements, except that cards are read from the on-line card reader designated as number 2.

PRINT Statement — PRINT n, List

In this statement, n is the statement number of a FORMAT statement, and the list is as described previously. The PRINT statement causes lines to be printed on the on-line printer designated as number 1. As many lines are printed as are necessary to use the amount of information specified in the list and contained in the FORMAT statement. The arrangement of information in the lines is defined in the FORMAT statement; each variable in the list is converted, also as defined in the FORMAT statement, and written on the printer. Up to 120 characters may be printed on one line.

PRINT ONE Statement — PRINT ONE n, List

This statement is exactly equivalent to the PRINT statement.

PRINT TWO Statement — PRINT TWO n, List

This statement is equivalent to the PRINT and PRINT ONE statements, except that lines are printed on the on-line printer designated as number 2.

PUNCH Statement — PUNCH n, List

This statement operates the same way as the PRINT statement, except, of course, that cards are punched on an on-line card punch instead of lines being printed. Up to 80 columns may be punched in one card.

PUNCH ONE Statement — PUNCH ONE n, List

This statement is exactly equivalent to the PUNCH statement.

PUNCH TWO Statement — PUNCH TWO n, List

This statement is equivalent to the PUNCH and PUNCH ONE statements, except that cards are punched on the on-line card punch designated as number 2.

READ INPUT TAPE Statement — READ INPUT TAPE i, n, List

This statement is used to read a magnetic tape which contains records of up to 80 Honeywell 800 characters in alphanumeric form. Such a tape may be produced in either of the following ways:

1. By the computer, operating in the parallel processing mode. A special program reads cards from an on-line card reader and writes the information onto a tape; no other parallel-processed program is affected nor slowed significantly. The tape so produced may later be read by an Algebraic Compiler program without ever removing the tape from the computer, or more often dismounted for use at a later time.
2. By an off-line card-to-tape converter.

These methods are often preferable to reading cards directly with a READ statement, because tape can be read much more rapidly than cards; if there is voluminous data, the difference in time can be appreciable.

In this statement, i is an unsigned fixed-point constant in the range of zero through 63, and must be the number of a magnetic tape unit which is available on the computer system to be used by the object program. Symbolic tape addresses are not permitted. The statement number of a FORMAT statement is given by n, and the list is as discussed previously.

With regard to end-of-file conditions, this statement operates much as the READ statement does, although the conditions detected are somewhat different. Either of the following is an end-of-file condition for the READ INPUT TAPE statement:

1. Detection of the physical end of the tape during the reading of a record. In this case, the reading of the record involved was completed, but there should not be any more valid information on the tape;
2. Detection of a record produced by a card which had the word FINIS punched in columns 2 through 6.

WRITE OUTPUT TAPE Statement — WRITE OUTPUT TAPE i, n, List

This statement is used to write a magnetic tape record containing up to 120 alphanumeric Honeywell 800 characters. Such a tape can then be printed by a parallel-processed program or an off-line tape-to-printer converter.

In this statement, i is an unsigned fixed-point constant in the range of zero through 63, and must be the number of a magnetic tape unit which is available on the computer system to be used by the object program. Symbolic tape addresses are not permitted. The statement number of a FORMAT statement is given by n, and the list is as discussed previously.

As many records are written as are required to exhaust the list. The FORMAT statement determines the type of conversion applied to each variable in the list. An END FILE statement should be given after writing the last record.

It may be noted that a tape to be printed, either by a parallel-processed program or by an off-line converter, must have information in each record to control the page spacing. See the discussion of the Hollerith field specification for a discussion of this topic.

With the WRITE OUTPUT TAPE statement, an end-of-file indication is given only by reaching the physical end of tape during the writing of the record. There is enough tape beyond the end-of-tape marker to allow continued writing of as many as 2,048 words.

READ TAPE Statement — READ TAPE i, List

This statement is used to read tapes produced by a WRITE TAPE statement, and is similar to that statement in all respects.

It is important to realize that the numbers read from tape are in no way associated with the names of the variables that were originally placed on the tape. Once the tape has been written with a WRITE TAPE statement, the only information on the tape consists of the variables themselves, not their names. Thus, if the list associated with the WRITE TAPE statement is A, B, C, and the list associated with the READ TAPE statement is C, A, B, the information formerly in A will be read back into C, etc.

The end-of-file condition for the READ TAPE statement consists of detecting the end-of-file indication written on a tape by an END FILE statement. Note that it is necessary to read the record produced by the END FILE statement in order to get the end-of-file indication; it is not given on reading the last record before the indication written by the END FILE statement.

WRITE TAPE Statement — WRITE TAPE i, List

This statement is used to write a magnetic tape which is to contain Honeywell 800 words exactly as they appear in storage, without any type of conversion; note that no FORMAT statement is referenced by the WRITE TAPE statement. It is used in problems where there is too much intermediate data to be stored within the computer; intermediate results can be written onto tape, then brought back in later with a READ TAPE

statement. A tape prepared by a WRITE TAPE statement cannot ordinarily be meaningfully printed on an off-line printer. In the statement, i is an unsigned fixed-point constant in the range of zero through 63, and must be the number of a magnetic tape unit which is available on the computer system to be used by the object program. Symbolic tape addresses are not permitted.

With any of the four preceding tape statements, there exists the possibility that there could be an error on the tape. It is expected that the incidence of tape errors with the Honeywell 800 System will be very small, and that most of the errors which do occur can be eliminated by re-reading the tape record or by use of Orthotronic correction. If, however, an uncorrectable error does occur, it is desirable to be able to alter the flow of control in the program. In such a rare case, it may be possible to go on to the next set of data after printing an indication of the bad tape record; perhaps it is necessary to stop the program if an uncorrectable error occurs. The alteration of the normal statement processing sequence in the event of such an error can be effected by use of the IF PARITY statement. This statement, which was discussed in Section IV on control statements, must be the next executable statement after the tape statement, if it is used. If the statements IF PARITY and IF END OF FILE are both used, as they often will be, the IF PARITY must be first.

END FILE Statement — END FILE i

This statement is used to write, onto magnetic tape number i, a signal which can be recognized by the IF END OF FILE statement for binary tapes and by the off-line printer for alphanumeric tapes. It is ordinarily used to indicate that there is no more valid information on the tape, but it may also be used to separate groups of records into files, for any purpose that may be convenient.

REWIND Statement — REWIND i

This statement is used to rewind, to the beginning of the tape, the magnetic tape reel mounted on tape unit number i.

BACKSPACE Statement — BACKSPACE i

This statement is used to backspace, by one logical record, the tape mounted on tape unit number i. In the case of a tape written by the WRITE OUTPUT TAPE statement, a logical record is the same as a physical record. In the case of a tape written by a WRITE TAPE statement, a logical record may be one or more physical records, depending on the

size of the logical record. If the tape is already at the beginning of the tape when this statement is executed, the tape will not move and no indication will be given.

BUFFER Statement — BUFFER (n_1, n_2, n_3), (m_1, m_2, m_3), ...

The BUFFER statement makes it possible to shorten considerably the execution time of a program involving the reading and writing of large arrays with the READ TAPE and WRITE TAPE statements. When such a statement is buffered, the reading or writing is carried on simultaneously with computation, so that there is very little time added to the program by the tape operations.

The statement may only be used in connection with the READ TAPE and WRITE TAPE statements, and the list in each case must consist of the name of exactly one array shown in non-subscripted form. The name of the array must naturally appear elsewhere in a DIMENSION statement. The symbols used in the specimen statement above are to be interpreted as follows:

n_1 = IN for reading, OUT for writing

n_2 = number of the tape unit involved

n_3 = number of words in the longest record to be read or written with this tape

As many buffers as required may be set up with one BUFFER statement, or separate statements may be used.

For input, a buffer area of the size specified (n_3) will be set up for each buffered tape, plus a transfer buffer of approximately 105 words. For output, there will again be a 105-word transfer buffer, plus one buffer area (no matter how many tapes are buffered) of a size equal to the largest buffer requested. Input buffering, if used, will result in the compilation of an object routine for handling the buffering of approximately 150 instructions, as will output buffering if used. Several additional special register groups in the Honeywell 800 are required for buffering.

The READ INPUT TAPE and WRITE OUTPUT TAPE statements may not be buffered.

ERASE Statement — ERASE (List)

This statement may be used to clear to zero the locations corresponding to the variables specified in the list. It is not, strictly speaking, an input or output statement, since it does not involve any input or output device. It is discussed here because it resembles an input or output statement to the extent that it does require a list. This is an executable statement, i. e., the locations mentioned are cleared to zero every time the statement is encountered in the object program.

As an example, an acceptable ERASE statement would be:
ERASE (TEMP, ALINE, NZERO, (BLINE(I), I = 1, 9, 2))

As an illustration of several of the input and output statements, we shall show a complete program to read two matrices from tape, multiply them, and prepare an output tape containing the product matrix.

Suppose that matrix A is to be multiplied by matrix B to give the product matrix C. Suppose that matrix A has L rows and M columns, B has M rows and N columns; matrix C will then have L rows and N columns. If $c_{i,k}$ is a typical element of the C matrix, it is defined by the summation formula:

$$c_{i,k} = \sum_{j=1}^{M} a_{i,j} \cdot b_{j,k}$$

This formula must then be evaluated for every combination of i and k, where i is between 1 and L, and k is between 1 and N.

Suppose now that the input tape has been prepared from cards which were punched as follows. On the first card, L is punched in columns 1 and 2, without a sign; similarly, M is punched in columns 11 and 12, and N in 21 and 22. The columns between the numbers are blank. The elements of matrix A then follow on the second and succeeding cards, as many as are required, in correct order, i. e., with the first subscript varying most rapidly. Each element is punched in 10 columns, with no space between elements, in the general form ±nn.nnnEee. That is, the numbers are punched with a decimal point (which actually may be anywhere in the number), and with an exponent. The first element of matrix B appears immediately after the last element of matrix A. This is not necessarily the best way to set up the input cards, especially if the matrices are very large, because of the possibility of error in punching the numbers or of getting the cards out of sequence, and because of the inflexibility of the scheme. For these reasons, one might prefer to punch one element on each card, with an identification of which element each one is, as was done in a previous example.

After the two matrices have been multiplied, the matrix C is to be written out on an output tape for subsequent printing, with the following page format. At the top of the page there is to be a heading line which identifies the output as being the product matrix C, and the values of L, M and N. The elements are then to be printed, in normal array order, one to a line. Each number is to be converted by the "E" field specification, in the same form as the input elements except that they are printed one to a line for ease of readability.

Furthermore, each element is to be preceded by an identification of which element it is, so that a typical element would appear as:

C(12, 19)= -78.914E 02

A scale factor of 2 will be required to move the decimal point of the number two places to the right of its normal position with the "E" field specification.

No IF END OF FILE statement is used, since it is assumed that the tape is long enough to hold all the elements and there would be no end-of-file indication otherwise. The IF PARITY statement transfers control to a STOP statement if an uncorrectable error is found.

The DO loop below which performs the matrix multiplication contains three DO's. The innermost DO performs the summation shown in the formula. The middle DO moves through the columns of the B matrix, and the outer DO moves through the rows of the A matrix. Notice that since we accumulate the C element in the location assigned to the element, it is necessary to set the location to zero initially. The DIMENSION statement establishes storage space for the three matrices, and sets the maximum size of the matrices which can be handled, which is assumed to be 30 x 30 here.

ALGEBRAIC COMPILER STATEMENT

TITLE _____ WRITTEN BY _____ CHECKED BY _____ DATE _____ PAGE ___ OF ___

A, B, C	STATEMENT NUMBER	CONTINUATION	ALGEBRAIC COMPILER STATEMENT (1 / 11 / 23 / 38 / 52 / 66 / 72 / 80)
1	TITLE		MTX MPY
2			DIMENSION A(30,30), B(30,30), C(30,30)
3			READ INPUT TAPE 6, 10, L, M, N, ((A(I,J), I=1,L), J=1,M),
4		X	((B(J,K), J=1,M), K=1,N)
5			IF PARITY 8,9
6	8		STOP
7	10		FORMAT (I2, 8B, I2, 8B, I2/(8E10.0))
8	9		DO 11 I=1,L
9			DO 11 K=1,N
10			C(I,K) = 0.0
11			DO 11 J=1,M
12	11		C(I,K) = C(I,K) + A(I,J) * B(J,K)
13			WRITE OUTPUT TAPE 7, 12, L, M, N
14	12		FORMAT (25HPRODUCT MATRIX C, WITH L= I2, 4H M= I2, 4H N= I2///)
15			WRITE OUTPUT TAPE 7, 13, (((I, K, C(I,K)), I=1,L), K=1,N)
16	13		FORMAT (2HC(I2, 1H, I3, 2H)= 2PE15.3)
17			REWIND 6
18			END FILE 7
19			REWIND 7
20			STOP
21			END

A	STAT. NO. / DATA NAME	COMMAND CODE	A ADDRESS	B ADDRESS	C ADDRESS

SECTION VI

FUNCTIONS

General Considerations

The techniques described in this section are designed to provide a number of conveniences to the programmer.

1. Open functions and library functions to provide an easy way to obtain certain commonly-used operations without actually writing a set of statements to compute them.
2. All of the methods of this section, except open functions, provide a means to put a program in memory, in one place, then call it into operation from many other places in the program. This saves both programming effort and storage space.
3. The FUNCTION and SUBROUTINE statements provide a way to break a program into subprograms which may be compiled independently if desired. This makes it possible to compile and check out a complete program in sections, as it is written, and to recompile only the affected parts when corrections must be made.

Before examining each of the techniques in detail, a little must be said about the mechanics of compilation of a program written for the Honeywell Algebraic Compiler. It is not necessary to go into all the intricacies of operation of the Compiler, but in order to understand the various functions it is necessary to know something about the idea of the Collector Tape. The collector is a tape on which are "collected" all the compiled programs which are available for operation on the computer at an installation. The output of a compilation is a set of records added to the collector tape, plus an optional listing which shows the storage requirements of the program and certain other information. The output of a compilation, however, is not in final form on the collector tape, but rather is on the tape in sections, ready to be "collected" together to form a running program. The final collection is initiated by the computer operator, using control cards which specify what sections to collect to form a program and other important information which is described later.

The point of this apparent digression will become clear in the discussion of the form in which the various types of functions appear on the collector tape; it will also become clearer then why it is advantageous that the collector tape be set up as it is.

Open Functions

There are a number of operations, all of which could be programmed by writing suitable combinations of ordinary statements, which are required so commonly that they have been

provided as built-in functions in the Algebraic Compiler system. These open functions, as they are called, are compiled into the object program wherever their names appear. To emphasize: if an open function is used many times, it appears in the object program many times, and there is no question of setting up a mechanism for going to the function and then returning to the section of the program which called it into operation. This is the essence of an open function, that it is inserted wherever needed and as many times as needed. None of the other types of functions have this characteristic.

The Honeywell Algebraic Compiler, as supplied, contains 13 such open functions, with provision for adding several more, as described in the Operations Manual. The standard functions are shown in Figure 6.

Name	Mode of Argument	Mode of Function	Type of Function	Definition
ABSF XABSF	Floating Fixed	Floating Fixed	Absolute Value	$\lvert Arg \rvert$
INTF XINTF	Floating Floating	Floating Fixed	Truncation	Sign of Arg times Largest Integer $\leq \lvert Arg \rvert$
MODF XMODF	Floating Fixed	Floating Fixed	Remaindering (see note 1 below)	$Arg_1 \pmod{Arg_2}$
FLOATF	Fixed	Floating	Float	Float Fixed Number
XFIXF	Floating	Fixed	Fix	Same as XINTF
SIGNF XSIGNF	Floating Fixed	Floating Fixed	Transfer of Sign	Sign of Arg_2 times $\lvert Arg_1 \rvert$
DIMF XDIMF	Floating Fixed	Floating Fixed	Diminishing	$\lvert Arg_1 - Arg_2 \rvert$
EXCLORF	Boolean	Boolean	Exclusive OR	$(Arg_1 + Arg_2) * (-(Arg_1 * Arg_2))$

NOTES: 1. The function $MODF(Arg_1, Arg_2)$ is defined as $Arg_1 - \left[Arg_1 / Arg_2\right] Arg_2$, where $\left[x\right]$ = integral part of x.

Figure 6. Open Functions of Honeywell Algebraic Compiler

The name of an open function consists of four to seven alphabetic or numeric characters (but no special characters), of which the first must be alphabetic and the last F. The first character must be X if and only if the value of the function is to be fixed point. The name of the function is followed by parentheses enclosing the argument(s), which are separated by

commas if there is more than one. Each open function has a prescribed mode (fixed or floating point) for its argument(s) and for its value; different functions must be used for each combination of modes of argument(s) and function value. The output of an open function always consists of one value. Any expression (of the correct mode) including another function, may be used as an argument of an open function.

The absolute value function is a good example of an operation which could easily enough be programmed explicitly (using an IF statement). However, since it is required so frequently it is much more convenient for the programmer to use it as an open function. One common use for the absolute value function is in testing for completion of an iterative process which is to be done repeatedly until two successive results are the same to within some pre-established tolerance. Suppose that a loop has been set up to compute a value, that the current value is named CURRNT, that the previous value is named PREV, and that the tolerance has been read in from cards into TOLER. If the absolute value of the difference between CURRNT and PREV is less than TOLER, we want to go to statement 112, but if it is greater than or equal to TOLER, we want to go back to statement 77. One statement will do this:

```
IF(ABSF(CURRNT - PREV) - TOLER) 112, 77, 77
```

An alternative method would be to use the DIMF function, which gives the absolute value of the difference between its two arguments:

```
IF(DIMF(CURRNT, PREV) - TOLER) 112, 77, 77
```

There is little to choose between the two, in this case.

Library Functions

One of the characteristics of an open function is that it requires only a few instructions in the object program. The library functions are provided for the computation of functions which are also commonly used, but which require more instructions. These are used by the programmer in exactly the same way as open functions, but are treated differently by the Compiler. The most important difference is that a library function is inserted into the object program only once, no matter how many times the programmer uses it, even if it is used in different subprograms (see below). Thus, if the square root function (see Figure 7) is used 10 times in five different subprograms, it will still appear in the object program only once, for the use of all subprograms.

The mechanism by which this is done involves the collector tape. The library functions are on the collector tape, ready to be inserted when the object program is finally collected and run. When a compiled program is collected, the collection program in effect scans a

table which has been set up for each subprogram involved, and places in the final program all library functions which appear in any subprogram. No extra effort is required to be sure that library functions do not appear more than once in the object program.

The Honeywell Algebraic Compiler, as supplied, contains 15 library functions, which are listed in Figure 7. No provision is made for adding others, but the FUNCTION subprogram method is available for adding functions to the collector tape; such functions can then be used in much the same way as library functions.

Wherever the name of a library function appears in a program, control is transferred to the function program; at the end of the function program, there is a return linkage to get back to the place from which control was transferred. All of this is automatic, so far as the programmer is concerned; it is only necessary to write the name of the function for everything described above to happen.

Name	Type of Function	
LOGF	Logarithm to the base e	Floating Point Functions
*SINF	Sine	
*COSF	Cosine	
EXPF	Exponential (e^{arg})	
SQRTF	Square Root	
*ATANF	Arctangent	
*TANHF	Hyperbolic Tangent	
**MAX0F	Choosing Largest Value (fixed argument)	
MAX1F	Choosing Largest Value (floating argument)	
***MIN0F	Choosing Smallest Value (fixed argument)	
MIN1F	Choosing Smallest Value (floating argument)	

The MAX and MIN functions may be preceded by an X to indicate that the resultant value is to be fixed point.

*All trigonometric functions deal with angles in radians.

**MAX (Arg_1, Arg_2, ...)

***MIN (Arg_1, Arg_2, ...)

Figure 7. Library Functions of Honeywell Algebraic Compiler

The name of a library function consists of four to seven alphabetic or numeric characters (but no special characters), of which the first must be alphabetic and the last F. The first character is X if and only if the value of the function is fixed point. The name of the function is followed by parentheses enclosing the argument(s), which are separated by commas if there is more than one. Each library function has a prescribed mode (floating point) for its

argument(s) and for its value; floating-point functions require floating-point arguments except for MAX and MIN functions which are defined by combinations. Any expression (of the correct mode), including another function, may be used as an argument of a library function. The output of a library function is always one value.

As an elementary example, a computation done before may now be carried out using the square root function. The problem was to compute:

$$D = \sqrt{(X2 - X1)^2 + (Y2 - Y1)^2}$$

This can be done with the statement:

```
D = SQRTF((X2 - X1) ** 2 + (Y2 - Y1) ** 2)
```

Incidentally, the use of the square root function is somewhat to be preferred over raising to the 0.5 power, since the square root function operates a little more rapidly.

As another example, suppose that it is required to compute, for a given value of X already in storage, the value of the following function:

$$Z = -X \operatorname{ctn} X + \log |\sin X|$$

The logarithm here is the natural logarithm, which is what is supplied with the system. The system does not include the cotangent function, so we shall compute it from:

$$\operatorname{ctn} X = \cos X / \sin X$$

The following statement will compute the value of Z:

```
Z = -X *  COSF(X) / SINF(X) + LOGF(ABSF(SINF(X)))
```

It may be noted in passing that this statement contains the minimum possible number of parentheses.

Another example of the same general nature but illustrating how expressions may be used as arguments, is the evaluation of the following formula:

$$ZINT = \frac{1}{C\sqrt{A \cdot B}} \arctan\left(e^{CX}\sqrt{\frac{A}{B}}\right)$$

This may be computed by use of the following statement:

ALGEBRAIC COMPILER STATEMENT

TITLE ☐☐☐☐☐☐☐☐ WRITTEN BY _____ CHECKED BY _____ DATE _____ PAGE ___ OF ___

A, B, C	STATEMENT NUMBER	CONTIN	ALGEBRAIC COMPILER STATEMENT	
1		6	11 23 38 52 66 72	80
			ZINT = (1.0/(C * SQRTF(A*B))) * ATANF(EXPF(C*X) * SQRTF(A/B))	

This statement contains one extra set of parentheses: those enclosing the multiplier of the arctangent were added for clarity. Incidentally, the formula might have been written as:

$$ZINT = \frac{\arctan\left(e^{CX}\sqrt{\frac{A}{B}}\right)}{C\sqrt{A \cdot B}}$$

and the statement written as:

```
ZINT = ATANF(EXPF(C * X) * SQRTF(A / B)) / (C * SQRTF(A * B))
```

The result is, of course, the same. (All the parentheses here are necessary.)

Defined Functions

The open and library functions discussed so far are both available in the system as supplied, and are brought into the object program simply by writing their names. It frequently happens that the programmer finds some computation occurring many times in a program, so that it would be convenient to be able to define the computation as a function for the purpose of the one program alone, and then use it as often as required in that program. This is how a defined function is used.

A defined function, also called an arithmetic statement function, is defined with a single statement and then brought into operation elsewhere in the source program, wherever its name appears. A defined function applies only to the program or subprogram in which it appears.

A defined function is defined to the Compiler by a statement of the form a = b, where a is the function name and b is an expression. The name of a defined function consists of from four to seven alphabetic or numeric characters (but no special characters), of which the first must be alphabetic and the last F. The first character must be X if and only if the value of the function is to be fixed point. The name of the function is followed by parentheses enclosing the argument(s), which are separated by commas if there is more than one. In the definition statement, the arguments must be distinct non-subscripted variables; there may be any number of them from one to 40. The right-hand side of the definition statement may be any expression which does not involve subscripted variables; it may involve variables not specified as arguments, and may make free use of other functions. The arguments which appear in the definition statement are only dummies which specify to the Compiler how to substitute into the defined function the arguments which are written when the defined function is later used. Therefore, the variable names used in the function definition are unimportant, except as they indicate fixed or floating point, and may be the same as the names of variables appearing elsewhere in the program.

So far we have spoken only of the definition of a defined function. In order to use a defined function, one merely writes the name of the function whenever its value is wanted, writing for arguments any expressions which agree in number, order, and mode, with the arguments as stated in the definition of the function. These (actual) arguments may be subscripted, whereas in the definition the dummy arguments cannot. The output of a defined function always consists of one value.

The program which is compiled to carry out the operations specified in the function definition statement appears once in the object program, at the end of the program in which it appears (but recall that the definition applies only to the program or subprogram in which it appears). Each time the defined function is used (by writing its name with suitable arguments), the object program then refers to the one place where the defined function appears. A defined function is thus compiled as a closed subroutine.

For a typical example, suppose that in a certain problem it is frequently necessary to evaluate the formula:

```
(-B + SQRTF(B ** 2 - 4. * A * C)) / (2. * A)
```

Each time this formula is evaluated, it is necessary to use different expressions for A, B, and C. The function could be defined by the statement:

```
ROOTF(A, B, C) = (-B + SQRTF(B ** 2 - 4. * A * C)) / (2. * A)
```

Now suppose that it is necessary to evaluate this formula with A equal to DATA (6), B equal to 12.8, and C equal to the absolute value of X minus Y. The result is to be added to Z^3 and the sum stored as VALUE. The following statement accomplishes this:

```
VALUE = ROOTF(DATA(6), 12.8, ABSF(X - Y)) + Z ** 3
```

It would have been possible, although pointless, to do the same thing with these four statements:

ALGEBRAIC COMPILER STATEMENT

TITLE _____ WRITTEN BY _____ CHECKED BY _____ DATE _____ PAGE ___ OF ___

A, B, C	STATEMENT NUMBER	CONTIN	ALGEBRAIC COMPILER STATEMENT (1 6 11 23 38 52 66 72 80)
1			A = DATA(6)
2			B = 12.8
3			C = ABSF(X-Y)
4			VALUE = ROOTF(A, B, C) + Z**3

This alternative is shown to try to bring out the point that the variables used in defining the function and the variables used for arguments in using the function are unrelated and independent. This point is seen even more clearly if we write the program in the following way,

which gives exactly the same result as the previous two:

ALGEBRAIC COMPILER STATEMENT

TITLE _____ WRITTEN BY _____ CHECKED BY _____ DATE _____ PAGE ___ OF ___

A, B, C	STATEMENT NUMBER	CONTIN 6	ALGEBRAIC COMPILER STATEMENT (1 11 23 38 52 66 72 80)
1			A = 12.8
2			B = ABSF(X-Y)
3			C = DATA(6)
4			VALUE = ROOTF(C, A, B) + Z**3

Suppose that at some other point in the program we need to evaluate the formula with A equal to the square root of X^2 plus Y^3, B equal to 12.8, and C equal to X + Y + Z all divided by 6; the result is to be raised to the 1.789 power and stored as the new value of HEAT:

```
HEAT = (ROOTF(SQRTF(X ** 2 + Y ** 3), 12.8, (X + Y + Z) / 6.)) ** 1.789
```

Two defined functions are involved in the following example, both involving only the one variable X; the other variables are defined elsewhere in the program. The indefinite integral of the function:

$$\frac{1}{(AX^2 + BX + C)^{3/2}}$$

is given by:

$$\frac{4AX + 2B}{(4AC - B^2)\sqrt{AX^2 + BX + C}}$$

This expression is to be evaluated; this could be done with one defined function. Here, however, we shall use two defined functions in order to make the first one available for use by itself elsewhere in the program. The definitions:

ALGEBRAIC COMPILER STATEMENT

TITLE _____ WRITTEN BY _____ CHECKED BY _____ DATE _____ PAGE ___ OF ___

A, B, C	STATEMENT NUMBER	CONTIN 6	ALGEBRAIC COMPILER STATEMENT (1 11 23 38 52 66 72 80)
1			FXXF(X) = SQRTF(A*X**2 + B*X + C)
2			PNTF(X) = (4.*A*X + 2.*B)/((4.*A*C - B**2) * FXXF(X))

If now we want to obtain the value of the integral evaluated between the lower limit 1.0 and the upper limit X, this value to be named FCTN, we can write:

```
FCTN = PNTF(X) - PNTF(1.0)
```

The X here must of course be defined elsewhere in the program; it has nothing to do with the X used in the definitions.

FUNCTION Subprograms

The FUNCTION statement described below may be viewed in two rather different ways. One is to regard it as a convenient way to set up a function to carry out some often-used segment of a program (similar to a defined function) using as many statements as may be required, however, instead of the one statement to which a defined function definition is limited. The other is to regard a FUNCTION subprogram as an alternative way to do approximately the same thing that a SUBROUTINE subprogram (see below) does, i.e., allow one to partition a complete program into pieces which may be compiled independently.

A FUNCTION subprogram is one which is defined by the use of a FUNCTION statement followed by any number of statements, and then activated elsewhere in the program by writing the name of the function with suitable arguments. A FUNCTION subprogram is an independent part of a total program. Its variable names may be the same as names which appear in the main program or in other subprograms. It may have its own DIMENSION and EQUIVALENCE statements. Any defined functions appearing in a FUNCTION subprogram apply only to that subprogram. The arguments in a FUNCTION statement may be the names of arrays as well as the names of single variables. The output of a FUNCTION subprogram always consists of a single value. A FUNCTION subprogram may be batch compiled with a main program and/or other subprograms, or it may be compiled independently.

The name of the FUNCTION subprogram must not be one that is already on the collector tape. FUNCTIONs on the collector include the names of the library functions, and in some cases these same names preceded by a Q. The user is urged to check against the list of functions already on the collector tape at his installation.

FUNCTION Statement FUNCTION Name $(a_1, a_2, \ldots, a_n)$

The name of a FUNCTION subprogram consists of from one to six alphabetic or numeric characters (but no special characters), the first of which must be alphabetic; the first character must be I, J, K, L, M, or N if and only if the value of the function is to be fixed point, and the last character must not be F if the name is more than three characters long. (Notice the contrast in naming between this and the functions above.) The name must not appear in a DIMENSION statement in the FUNCTION subprogram, nor in a DIMENSION statement in any program which uses the subprogram. (Otherwise the FUNCTION will be mistaken for a subscripted variable.) The name must appear at least once in the FUNCTION subprogram as a variable on the left-hand side of an arithmetic statement, or alternatively in an input statement list. The name of the FUNCTION subprogram is followed by parentheses enclosing the argument(s), which are separated by commas if there is more than one. In the FUNCTION

statement, the arguments must be distinct non-subscripted variables appearing on the right-hand sides of executable statements of the subprogram. There may be any number of arguments from one to 48 (see Appendix C).

As with a defined function, the variables appearing as arguments in the FUNCTION statement are only dummies, and the remarks about dummy variables made above also apply here. In addition, none of the dummy variables of a FUNCTION subprogram may appear in EQUIVALENCE or COMMON statements in the subprogram.

The FUNCTION statement must be the first statement of the subprogram; all statements which follow, up to the END statement which must appear at the physical end of the subprogram, are taken to be part of the FUNCTION subprogram. The FUNCTION subprogram may use any type of statement except SUBROUTINE or another FUNCTION. Although FUNCTION and SUBROUTINE statements may not appear in a SUBROUTINE subprogram, i.e., it is not possible to define other subprograms within a subprogram, there is no restriction against using other subprograms within a subprogram, except that a subprogram may not use another subprogram of a higher level and may not use itself. If a COMMON statement is used in the subprogram, it naturally refers to the one common storage area which is the same for all programs collected together. This provides a means of establishing the correspondence between variables in the subprogram and variables in the main program or in other subprograms. This correspondence does not exist otherwise; remember that every subprogram is an independent entity. The dummy variables used as arguments in the FUNCTION statement must appear in non-subscripted form in the FUNCTION statement, but there is no such restriction on any variables in the subprogram. A FUNCTION subprogram must contain at least one RETURN statement, in order to set up the linkage back to the calling program.

All of the above applies only to the definition of a FUNCTION subprogram. To use such a subprogram, it is only necessary to write the name of the function with arguments which agree in number, order, and mode, with those in the FUNCTION statement. Furthermore, when a dummy argument is the name of an array, the corresponding actual argument must also be an array. The dummy array name must appear in a DIMENSION statement in the subprogram, and the actual array name must appear in a DIMENSION statement in the program requesting the FUNCTION subprogram. The dimensions for each must be the same. Dummy variables which represent single variables may be replaced with any expressions of the correct mode, including subscripted variables, constants, other functions, etc.

The object program which is compiled to carry out the operations specified in a FUNCTION subprogram will appear in the object program once, no matter how many times the subprogram is used.

For a first example, assume a number of two-dimensional arrays with maximum dimension of 10 in each direction. Find the largest element (in absolute value) in a specified row of the array. The row number is identified by the dummy variable I, the actual number of rows and columns is N (which might be less than the maximum of 10, of course), and the dummy name of the array is A. The following FUNCTION subprogram would find the absolute value of the largest of the N elements in the Ith row of A:

ALGEBRAIC COMPILER STATEMENT

TITLE ☐☐☐☐☐☐☐☐ WRITTEN BY _____ CHECKED BY _____ DATE _____ PAGE ___ OF ___

A, B, C	STATEMENT NUMBER	CONTIN	ALGEBRAIC COMPILER STATEMENT
			FUNCTION BIGIN(A, I, N)
			DIMENSION A(10,10)
			BIGIN = 0.0
			DO 90 J=1,N
			IF(ABSF(A(I,J)) - BIGIN) 90, 90, 89
	89		BIGIN = ABSF(A(I,J))
	90		CONTINUE
			RETURN
			END

Now, whenever, the element with the largest absolute value in a given row of a matrix of maximum dimension 10 x 10 is needed, it can be obtained simply by writing BIGIN with suitable arguments.

As an example, suppose that we want to divide every element of the first row of the matrix DATA by the absolute value of the largest element in the first row (largest in absolute value), this can be done.

ALGEBRAIC COMPILER STATEMENT

TITLE ☐☐☐☐☐☐☐☐ WRITTEN BY _____ CHECKED BY _____ DATE _____ PAGE ___ OF ___

A, B, C	STATEMENT NUMBER	CONTIN	ALGEBRAIC COMPILER STATEMENT
			DIMENSION DATA (10, 10)
			BIG =1.0/(BIGIN(DATA,1,N))
			DO 700 J = 1, N
	700		DATA(1,J) = DATA(1, J) * BIG

We would sometimes like to know the column number of the largest element, in order to do some interchanging of columns; this would require the FUNCTION subprogram to return to us a fixed-point variable as well as a floating-point variable. If we needed only the column number and not the element itself, the FUNCTION subprogram could easily be modified to give this result, but if we need both the largest element and the column number, we cannot use the FUNCTION method since there is no way to get more than one value as an output. This, however, can be done with the SUBROUTINE subprogram, although in a slightly different way, as we shall see below.

The example demonstrated the use of the FUNCTION subprogram to do something that cannot be done with a defined function, because it required more than one statement and because it involved an array. The next example is based on a situation involving no arrays, but requiring more than one statement. The function shown below has three different formulas for its indefinite integral, depending on the relative sizes of the parameters A and B. If we had to calculate this integral very often, we would want very much to make a function out of it, but it would require the use of something other than a defined function since it cannot be done in one statement.

$$\int \frac{dX}{A^2 - B^2 \sin^2 X} = \frac{1}{A\sqrt{A^2 - B^2}} \arctan \frac{\sqrt{A^2 - B^2}\tan X}{A}, \quad A^2 > B^2$$

$$= \frac{1}{A^2} \tan X, \quad A^2 = B^2$$

$$= \frac{1}{2A\sqrt{B^2 - A^2}} \log \left| \frac{\sqrt{B^2 - A^2}\tan X + A}{\sqrt{B^2 - A^2}\tan X - A} \right|, \quad B^2 > A^2$$

This will also give some further practice in writing moderately complex statements involving open and library functions, as well as in the use of the FUNCTION statement and subprogram.

ALGEBRAIC COMPILER STATEMENT

TITLE _____ WRITTEN BY _____ CHECKED BY _____ DATE _____ PAGE ___ OF ___

	A, B, C	STATEMENT NUMBER	CONTIN	ALGEBRAIC COMPILER STATEMENT
1				FUNCTION SININT (A, B, X)
2				ROOT = SQRTF(ABSF(A**2-B**2))
3				TANX = SINF(X) / COSF(X)
4				IF(B**2 - A**2) 400, 500, 600
5		400		SININT = (1./(A*ROOT)) * ATANF(ROOT*TANX/A)
6				RETURN
7		500		SININT = TANX/ A**2
8				RETURN
9		600		SININT = (1./(2.*A*ROOT))*LOGF(ABSF((ROOT*TANX+A)/(ROOT*TANX-A)))
10				RETURN
11				END

The shortcuts taken here are worth noticing. Since the square root of $A^2 - B^2$ or $B^2 - A^2$ occurred several times, the square root of $A^2 - B^2$ was computed before going into the test to determine the relative size of A^2 and B^2, so that the instructions for the computation would only have to appear once; this saves Compiler time, programming time, and object program memory space. The same type of precomputation was used to get the tangent of X. There are three RETURN statements above; it would have been acceptable, although a trifle longer, to place one RETURN at the end and transfer to it from the other places with GO TO's.

Now if it should be desired to find the value of this integral for A equal to π and B equal to B(5), evaluated between the lower limit of X1 and the upper limit of X2, and store the integral as VALUE, it could be done with:

VALUE = SININT(3.14159, B(5), X2) - SININT(3.14159, B(5), X1)

SUBROUTINE Subprograms

As with FUNCTION subprograms, SUBROUTINE subprograms may be regarded in two different ways: either as a way to do certain things that cannot be done with defined functions or with FUNCTION subprograms; or as a way to break a complete program into parts which can be compiled and checked out separately if desired. As before, both interpretations lead to exactly the same statements in the source program, but once again it may clarify matters

to have the two different aspects in mind in reading the descriptions below. Whenever it is necessary for a subprogram to have more than one variable as output, the SUBROUTINE method must be used, because all of the other techniques permit only one output value. As a matter of actual usage, the SUBROUTINE subprogram is used more often for the purpose of partitioning a program. One reason for doing so is to allow for easy recompilation of parts that must later be modified or corrected. This saves the computer time of recompiling the parts that do not require changes.

A most important aspect of the SUBROUTINE method is that it is possible for a very large project to be divided into convenient parts which can be programmed, compiled, and checked out independently by different programmers. After all the parts have been finished, they can be compiled into a complete program and used.

Finally, there is another important usage of the SUBROUTINE technique. That is in the case where a program is too large to fit in storage at one time. It is possible, by use of control cards, to specify that two or more subprograms are to occupy the same locations in storage; this is called overlaying. When this is done, the effect of the CALL statement (see below) is to transfer control to the subprogram if it is already in storage, or to bring it in from tape and then transfer control to it if it is not already in storage.

In order to do all of this, it is necessary for the SUBROUTINE subprogram to have, when applicable, its own DIMENSION, EQUIVALENCE, and COMMON statements. And to emphasize the point once again, a SUBROUTINE subprogram must be regarded as an independent entity, regardless of whether it is being used for the purpose of partitioning or not.

SUBROUTINE Statement SUBROUTINE Name $(a_1, a_2, \ldots, a_n)$

The name of a SUBROUTINE subprogram consists of one to six alphabetic or numeric characters (but no special characters), the first of which must be alphabetic and the last must not be F if the name is more than three characters long. (Note that there is no requirement about the first character being I, J, K, L, M, or N to specify fixed point, since the mode of the arguments determines whether the results are fixed or floating-point variables.) The name must not appear in a DIMENSION statement in the SUBROUTINE subprogram, nor in a DIMENSION statement in any program which uses the subprogram. The name of the subroutine must not duplicate any name already on the collector tape. The name of the subprogram is followed by parentheses enclosing the argument(s), if any, which are separated by commas if there is more than one; if there are no arguments, parentheses are not required.

In the SUBROUTINE statement, the arguments must be distinct non-subscripted variables appearing in executable statements in the subprogram; input arguments must appear in the right-hand sides of statements or in lists of input statements, and output arguments in the left-hand sides of statements or in the lists of output statements. There may be any number of arguments up to 48, or none; in the latter case, COMMON statements would be used to establish correspondence between variables in this subprogram and in other subprograms or in the main program.

As with defined functions and FUNCTION subprograms, the arguments appearing in the SUBROUTINE statement are only dummies which, in this case, specify to the Compiler how to substitute into the subprogram the arguments which are written in the CALL statement when the subprogram is used elsewhere in the program. As usual, then, the variable and array names used as arguments in the SUBROUTINE statements are unimportant, except as they specify fixed or floating point, and may be the same as names appearing in the main program or in other subprograms. However, none of the dummy variables of a SUBROUTINE subprogram may appear in EQUIVALENCE or COMMON statements in the subprogram.

The SUBROUTINE statement must be the first statement of the subprogram. All statements from there to the END statement, which must be physically the last statement of the subprogram, are taken to comprise the subprogram. The SUBROUTINE subprogram may use any type of statement except a FUNCTION statement or another SUBROUTINE statement. (A SUBROUTINE subprogram may, however, call other subprograms and use FUNCTIONS.) If a COMMON statement is used in the subprogram, it of course refers to the one common storage area which is the same for all programs which are collected together. As noted above, this provides a way to establish correspondence between the names of variables in different subprograms and the main program. It must always be kept clearly in mind that such a correspondence does not exist otherwise; the name DATA in a subprogram is totally unrelated to the name DATA in the main program, unless a correspondence has been established by COMMON statements in both places. (See the complete discussion of the COMMON statement in Section VII.)

The dummy variables which are used in the SUBROUTINE statement must appear in non-subscripted form, but there is no such restriction on any of the variables in the subprogram itself, dummy or otherwise, nor on the arguments in the CALL statement. Free use may be made of expressions, including any type of function. A SUBROUTINE subprogram must contain at least one RETURN statement.

The object program which is compiled to carry out the operations specified in the SUBROUTINE subprogram will appear only once in the object program which calls the SUBROUTINE, regardless of how many times the subprogram is called. Each time the subprogram is used (by calling it with a CALL statement), the object program transfers control to the subprogram; the RETURN statement then transfers control back to wherever the subprogram was called from. A SUBROUTINE subprogram is thus compiled as a closed subroutine. A SUBROUTINE subprogram must not be written between two statements of another program. A SUBROUTINE subprogram may be batch compiled with a main program and/or other subprograms, or it may be compiled independently.

CALL Statement CALL Name (a_1, a_2, ..., a_n)

All of the above has to do with the SUBROUTINE statement and the statements which follow it, i.e., with the definition of the subprogram. In order to call the subprogram into operation, it is necessary to use the CALL statement, which transfers control to the subprogram and transmits the input arguments to it, and then transmits the output variables back to the calling program when the subprogram has been executed. It should be noted that a CALL statement may appear in a subprogram, i.e., one subprogram may call another, to any depth; this applies equally to FUNCTION and SUBROUTINE subprograms. A program may not call another program of a higher level, and a program may not call itself.

The arguments in the CALL statement must agree with those in the SUBROUTINE statement in number, order, and mode. Furthermore, if an argument in the SUBROUTINE statement is an array name, the corresponding argument in the CALL statement must be an array name. The dummy array name in the SUBROUTINE statement must appear in a DIMENSION statement in the subprogram, the array name in the CALL statement must appear in a DIMENSION statement in the calling program, and the dimensions must be the same. Dummy variables which represent single variables may be replaced with any expressions, including subscripted or non-subscripted variables, constants, other functions, etc. Literal alphabetic or numeric characters may not be used but, of course, alphanumeric variables may be used to carry alphabetic information to the subprogram.

When it is desired to use overlaying, i.e., to have two or more subprograms occupy the same locations in memory, the subprograms must be designated for overlaying at collection time. This is done with an OVERLAY control card, the details of which are described in the Operations Manual; when a subprogram is named on an OVERLAY card, its object program is set up a little differently and provision is made for reading it from the program tape when it is called. Then, the operation of the CALL statement is: transfer to the named subprogram

if it is already in storage; if not, bring it in from tape and then transfer to it. The programmer designates at collection time the subprograms which are to be overlayed; all the rest is automatic with the CALL statement.

When different people are working on parts of a very large program, it often happens that one person needs to compile his section without compiling all the other parts. This leads to a problem; in the section being compiled, there may be statements referring to other sections not being compiled at the time. Ordinarily, this leads to a diagnostic error indication and the compilation is not completed. Here, however, it is necessary to go ahead; presumably the programmer has made plans to get around the missing sections. In such a case, the missing subprograms must be named on NEGLECT control cards at collection time; the Compiler will then insert dummy RETURN statements, and the compilation and checkout can proceed.

RETURN Statement

RETURN

This statement terminates the execution of any FUNCTION or SUBROUTINE subprogram, and returns control to the calling program. A RETURN statement must, therefore, be the last-executed statement of every subprogram. It need not, however, be physically the last statement of a subprogram (or rather, next to the last; the END statement must be the last). A RETURN statement may appear at any point in a subprogram, and there may be any number of RETURN statements.

As a simple example of how a SUBROUTINE statement can be used as a more powerful version of a defined function, consider the example discussed in connection with the FUNCTION subprogram, in which the largest element of the Ith row of a matrix (largest in absolute value) and its column number is needed. This makes two output numbers, which means that a SUBROUTINE subprogram must be used, which could be as follows:

ALGEBRAIC COMPILER STATEMENT

TITLE _____ WRITTEN BY _____ CHECKED BY _____ DATE _____ PAGE ___ OF ___

A, B, C	STATEMENT NUMBER	CONTIN	ALGEBRAIC COMPILER STATEMENT (1 6 11 23 38 52 66 72 80)
1			SUBROUTINE BIGCOL (A, I, N, BIG, ICOL)
2			DIMENSION A(10,10)
3			BIG = 0.0
4			ICOL = 1
5			DO 900 J = 1, N
6			IF (ABSF(A(I,J)) - BIG) 900, 900, 1000
7	1000		BIG = ABSF (A(I,J))

```
8          ICOL = J
9     900  CONTINUE
10         BIG = A (I, ICOL)
11         RETURN
12         END
```

Now suppose that we have a matrix PROBD, that we want to find the largest element in the first row and store it as AMAX, then interchange the first column with whichever column contains the largest element in the first row. This can now be done with a few statements, as follows:

ALGEBRAIC COMPILER STATEMENT

TITLE _____ WRITTEN BY _____ CHECKED BY _____ DATE _____ PAGE ___ OF ___

	A, B, C	STATEMENT NUMBER	CONTIN	ALGEBRAIC COMPILER STATEMENT
1				CALL BIGCOL (PROBD, 1, N, AMAX, J)
2				DIMENSION PROBD(10,10)
3				DO 1200 I=1,N
4				TEMP = PROBD(I,1)
5				PROBD(I,1) = PROBD(I,J)
6		1200		PROBD(I,J) = TEMP

The CALL statement calls into operation the subprogram, and stores the values of AMAX and J which it computes. This is done only when the CALL statement is executed, not when the variables AMAX and J are used. There can be no ambiguity between the use of the J as a CALL argument and its use in the SUBROUTINE subprogram; recall that there is no correspondence between variables unless we establish it. We need do nothing more about the AMAX in order to get it stored; that is part of the combined operation of the CALL statement and the subprogram. The DO loop takes one row at a time, first moving the element in the first column to temporary storage, then moving the element in the Jth column to the first column, then moving the element in temporary storage to the Jth column. Note that this works properly even if the largest element should happen to be in the first column already; an IF statement at the start could be used to save the waste motions in this event.

Another example of the use of the SUBROUTINE subprogram appears in the next section, where we present a complete example, showing how to segment a program.

Summary of the Differences Between the Five Types of Functions

The Honeywell Algebraic Compiler has provision for five types of functions: open, library, defined, and those established by FUNCTION and SUBROUTINE statements. The

following summary shows the major differences between the five types.

Naming

The names of open, library, and defined functions are: four to seven alphabetic or numeric characters (but no special characters), the first of which must be alphabetic and the last F; the first character must be X if and only if the value of the function is to be fixed point. The name of a FUNCTION subprogram is: one to six alphabetic or numeric characters (but no special characters), the first of which must be alphabetic; the first character must be I, J, K, L, M, or N if and only if the value of the function is to be fixed point, and the last character must not be F if the name is more than three characters in length. The name of a SUBROUTINE subprogram is: one to six alphabetic or numeric characters (but no special characters), the first of which must be alphabetic and the last of which must not be F if the name is more than three characters in length.

Definition

Open and library functions are provided with the system, although the system as supplied may be expanded with other open functions, as described in the Operations Manual. Defined functions are defined by writing a single definition statement. A FUNCTION subprogram is defined by any number of statements following a FUNCTION statement. A SUBROUTINE subprogram is defined by any number of statements following a SUBROUTINE statement.

How Requested

Open, library, defined, and FUNCTION functions are brought into operation by writing the name of the function in an expression where its value is desired. SUBROUTINE subprograms are requested with a CALL statement.

Open vs. Closed

Open functions are compiled as open subroutines, i.e., they are compiled into the program once for every time they are requested by name. All of the others are compiled as closed subroutines, i.e., they are compiled into the program only once, regardless of how many times they are requested; control is transferred to them and then back to the requesting program.

How Control is Returned to Calling Program

Not applicable to open functions. Control is automatically returned to the calling program from a library or defined function. Control is returned to the calling program from a FUNCTION or SUBROUTINE subprogram by a RETURN statement in the subprogram.

Number of Arguments

The number of arguments for open and library functions are specified for each function. Defined functions may have any number of arguments from one to 40. A FUNCTION statement may have any number of arguments from one to 48. A SUBROUTINE statement may have any number of arguments from none to 48.

Number of Outputs

A SUBROUTINE subprogram may have any number of outputs; all of the others give only one output value.

Separate Compilation

FUNCTION and SUBROUTINE subprograms may either be batch compiled with a main program and/or other subprograms, or they may be compiled independently. Open and defined functions are always compiled as parts of some larger program. Library functions are stored on the collector tape in pre-assembled form.

Dummy Variables in Definition

Variable names in the definition of a defined function, and in FUNCTION and SUBROUTINE statements, are dummies. The variable names used in requesting or calling any function must, of course, be the names of actual variables. In the case of these three functions, the variables must agree in number, order, and mode, with the dummy variables used in the definitions. Since no definition statement is required in the case of open and library functions, the discussion of dummy variables in such statements is not applicable.

SECTION VII

SPECIFICATION STATEMENTS

General Considerations

There are three statements in the Honeywell Algebraic Compiler which are used only to provide the Compiler with necessary information about the program being compiled, all having to do with the assignment of storage locations to variables, although in three rather different ways.

DIMENSION Statement

DIMENSION v, v, v, ...

The DIMENSION statement is used to specify to the Compiler the dimensions of arrays; every variable in a program which appears in subscripted form must appear in a DIMENSION statement. In the general form of the statement given above, v is the name of a variable with one, two, or three unsigned fixed-point constants in parentheses. For each variable, the subscripts in parentheses give the maximum size of the array, and storage space is set aside accordingly. The number of subscripts written also indicates to the Compiler whether the array is one, two, or three dimensional. Any number of subscripted variables may appear in one DIMENSION statement, separated by commas; there may be any number of DIMENSION statements in a program or subprogram. A DIMENSION statement applies only to the program or subprogram in which it appears, even if the names of two arrays in different subprograms are the same.

As a simple example, consider the following DIMENSION statement:

```
DIMENSION DATA(20, 15), IJK(60), DATAR(2, 10, 30)
```

This specifies that DATA is a two-dimensional array of floating-point numbers, with the maximum size of the first subscript being 20 and of the second being 15; IJK is a one-dimensional array of 60 fixed-point numbers; DATAR is a three-dimensional array of floating-point numbers, the maximum subscripts sizes being 2, 10, and 30. In the case of the array DATA, storage space is set aside for a two-dimensional array consisting of a total of 300 locations. In the program which uses DATA, one must never specify a first subscript for DATA larger than 20 nor a second subscript larger than 15. It should be emphasized that a specific memory location is reserved for each element of each array. It must not be argued that the 300 locations can be used to make up any array totalling 300 elements, such as 6 x 50 or 3 x 100. Each dimension applies strictly to the subscript in its position, and the memory assignments apply specifically to each individual element of an array of just the maximum size indicated.

It is still permissible, of course, to use arrays of less than the maximum size in any dimension, but this amounts only to not using some of the assigned locations, not to reassigning any of them.

A DIMENSION statement must not include the name of the program in which it appears, nor the name of any FUNCTION or SUBROUTINE subprogram which the program uses.

EQUIVALENCE Statement — EQUIVALENCE (a, b, c, ...), (d, e, f, ...), .

This statement makes it possible to do two things which on occasion are very useful:

1. Assign two or more variables to the same storage location, where the logic of the program permits it, thus making possible significant reductions in storage space required;
2. Establish two or more names as synonyms for the same variable.

An EQUIVALENCE statement applies only to the program or subprogram in which it appears. It may be placed anywhere in a program or subprogram.

The variables within a set of parentheses, which may be subscripted with a single unsigned fixed-point constant, are assigned to the same location. There may be any number of variables within one set of parentheses, and any number of parentheses. Variables and arrays which are not mentioned in EQUIVALENCE statements are assigned to unique locations. Locations can be shared only among variables, not constants.

The meaning of a subscript in an EQUIVALENCE statement is different from its meaning in other statements. The meaning of C(p) in an EQUIVALENCE statement is: the (p - 1)th location after the one containing C, or, if C is an array, the (p - 1)th location after the one containing C(1), C(i, 1), or C(1, 1, 1). Since there is no zeroth element in an array, p must be greater than zero.

The simplest example is an EQUIVALENCE statement not involving arrays or subscripts. EQUIVALENCE (DATA, X, Z), where none of the variables is an array, would mean to assign the variables DATA, X, and Z to the same storage location. In order to do this, it is, of course, necessary to know that the program never stores a new value of any of these variables unless the old value in the location is no longer needed. This is the programmer's responsibility; if a new value of DATA is stored in the one location at a time when it contains a value of X which will be needed later, the program will give incorrect results. Neither the Compiler nor the object program has any way of checking for this sort of error.

If, however, the reason for using the EQUIVALENCE statement is to establish two or more names as synonyms, then there is no question of avoiding overlapping usage; the variables named are all the same one, and the intention is to be sure they are all assigned to the same location. For instance, inexperienced programmers often do not realize how much care must be exercised in writing the letters I and O, in order to be able to distinguish them from the digits 1 and 0. Suppose that in a certain program heavy use has been made of the symbol PILOT, but that the programmer was very careless in his writing and is afraid the names may not all have been punched correctly. His problems can be solved, in this case, by writing:

```
EQUIVALENCE (PILOT, P1LOT, PIL0T, P1L0T)
```

Suppose now that three arrays are to be assigned to the same set of storage locations and, for simplicity, that they all have the same number of elements. If we write EQUIVALENCE (A, B, C), the result will be as desired. Note that it is not necessary to show subscripts in order to do this, although the same result would be obtained by writing EQUIVALENCE (A(1), B(1), C(1)). Two things about this example should be noted. First, it does not matter how many dimensions these arrays have. A corollary to this is that there is no requirement of any sort that the arrays have the same maximum dimensions or any other type of correspondence between elements. It would be perfectly acceptable for A to be a one-dimensional array with 300 elements, B to be a 20 x 15 two-dimensional array, and C to be a 10 x 3 x 10 three-dimensional array. Or, if all were two-dimensional arrays, one could be 3 x 50, a second 50 x 3, and a third 5 x 30. Neither is there any requirement that the arrays all have the same total number of elements. The EQUIVALENCE statement only places the starting points in correspondence.

The second thing to note is that only one subscript is given in an EQUIVALENCE statement, even when referring to two- and three-dimensional arrays. This is the point of the statement that C(p) refers to the (p - 1)th location after the location for C. Suppose for an example that there are three one-dimensional arrays E, F, and G of maximum size 20 each, and that they are to be assigned the same space as a 3 x 4 x 5 three-dimensional array. Specifically, it is desired to assign the first one-dimensional array, E, to the space occupied by the first 20 locations of the three-dimensional array, called Z; the second 20 locations of Z are to be the same as the locations for F, and the last 20 locations of Z are to be the same as the locations for G. The following statement establishes these equivalences:

```
EQUIVALENCE (E, Z), (F, Z(21)), (G, Z(41))
```

If it is desired to establish equivalences involving specific elements of arrays, it is necessary to know exactly how arrays are stored. This information has been given before, but may be reviewed here. The first element of an array, the one corresponding to C(1), C(1, 1), or C(1, 1, 1), is stored first, with other elements being stored in order after it (in successively higher numbered locations, incidentally). The elements are stored in such a way that the first subscript varies most rapidly and the last least rapidly. Thus, a 3 x 3 matrix A would be stored in the order A(1, 1), A(2, 1), A(3, 1), A(1, 2), A(2, 2), A(3, 2), A(1, 3), A(2, 3), A(3, 3).

If it is now desired to make the main diagonal elements of A correspond to the single variables R, S, and T, it can be done with:

```
EQUIVALENCE (A, R), (A(5), S), (A(9), T)
```

For this to work, A must appear in a DIMENSION statement as DIMENSION A(3, 3); otherwise, the diagonal elements are not the first, fifth, and ninth elements of the matrix. If A appears as DIMENSION A(4, 4) and the diagonal elements are to correspond to W, X, Y, and Z, the statement should be:

```
EQUIVALENCE (A, W), (A(6), X), (A(11), Y), (A(16), Z)
```

No attempt should be made to do anything which amounts to changing the way an array is stored. For instance, it is not possible to make the elements of a vector V correspond to the main diagonal elements of a 3 x 3 matrix by writing:

```
EQUIVALENCE (A, V), (A(5), V(2)), (A(9), V(3))
```

Such a statement, which gives impossible instructions to the Compiler, will produce a diagnostic statement and stop the compilation. In order for it to be accomplished, the elements of the vector would have to be stored in non-consecutive locations, which cannot be done.

In order to use the EQUIVALENCE statement, it is obviously necessary to arrange the program and the EQUIVALENCE entries so that no data is destroyed until it is no longer needed and, to do the planning properly, it is necessary to know which statements can cause new values of variables to be stored. Statements which store new values:

1. Arithmetic statements store a new value of the variable on the left-hand side of the statement;
2. Execution of ASSIGN i TO n stores a new value of n;
3. Execution of a DO always changes the value of the index of the DO;
4. Any input statement stores new values of the variables in the list; the ERASE statement should be considered as an input statement for this purpose;
5. Certain ARGUS statements;
6. A CALL statement with output arguments.

COMMON Statement — COMMON A, B, C,....

Ordinarily, i.e., in the absence of a COMMON statement, variables are assigned memory locations separately for each subprogram. If the main program has a variable named X and a subprogram has a variable named X, the two X's are essentially different and two separate memory locations are set up for the two of them. This is generally what is desired, but there are times when it is very convenient to be able to specify to the Compiler that a variable in one subprogram is the same as a variable in another subprogram (whether or not the two variables have the same name). This can be done by proper use of COMMON statements.

Variables which are named in COMMON statements are stored in a special section of storage which is set aside for storing COMMON variables, and this is the COMMON area which is the same for all subprograms which are to be collected to be run together. The variables named in COMMON statements are then assigned to the COMMON area in the order in which they appear. There are two different ways in which the COMMON statement may be viewed. For instance, if the statement:

```
COMMON X, Y, Z
```

appears in both the main program and in a subprogram, then X, Y, and Z in both programs would be assigned to the same locations in COMMON, and the variables would be the same for both programs. If, on the other hand, the main program has the statement:

```
COMMON X, Y, Z
```

and a subprogram has the statement:

```
COMMON A, B, C
```

then X and A are assigned to the first location in COMMON and become equivalent, B and Y are assigned to the second location in COMMON, and similarly for Z and C. The implication is that the two sets are not the same, but that both sets are never needed at the same time and therefore sharing of storage locations is feasible.

The EQUIVALENCE statement is capable of the same two interpretations, although ordinarily it is the second which is used. One way to view these two statements is that EQUIVALENCE establishes either identity or storage sharing of variables within a main program or subprogram, whereas COMMON establishes identity or storage sharing of variables among a main program and subprograms. The difference in the mechanism of the two, viz., the setting aside of a separate COMMON area, is dictated by considerations involving the internal operation of the compiler.

Internal operating considerations also dictate the following: the COMMON area is in actuality composed of one area for arrays and another area for single variables. This fact is of importance to the programmer when setting up COMMON statements involving both arrays and single variables. The operation of the Compiler is best illustrated with an example. Suppose a program contains the statement:

```
COMMON ARRAY1, X, Y, ARRAY2, Z
```

The array ARRAY1 will be assigned to the array part of the COMMON area, using as many locations as required by the dimensions in the DIMENSION statement which mentions ARRAY1. X and Y will be assigned to the first and second locations of the single variable area (if they are indeed single variables), ARRAY2 to the array area following the locations assigned to ARRAY1, and Z to the third location of the single variable area. Now suppose that in a subprogram there appears the statement:

```
COMMON ARRAY3, A, B, ARRAY4, C
```

These arrays and single variables will be assigned to the two COMMON areas in exactly the same way as in the previous statement. If ARRAY3 has the same total number of locations (from its DIMENSION statement) as ARRAY1, then the first element of ARRAY4 and the first element of ARRAY2 will be assigned to the same location in the array part of the COMMON area. Since the arrays and single variables are handled separately in COMMON, the relative order of arrays vs. single variables does not matter. The following statement would be equivalent to the second statement above:

```
COMMON A, ARRAY3, B, ARRAY4, C
```

The separate handling of arrays and single variables also leads to the need for a precaution. Suppose that in the main program there is the statement:

```
COMMON A, B, C, D, E, F
```

and that we need to make A in the main program correspond to X in a subprogram, and to make F correspond to Y. Since variables are assigned to locations in COMMON on a basis of the order in which they appear in the COMMON statement, it is necessary somehow to "fill out" the COMMON statement in the subprogram, so that Y is the sixth variable. The only way to do this is to make up the names of variables which do not really exist in the subprogram:

```
COMMON X, D1, D2, D3, D4, Y
```

One might be tempted to make up a one-dimensional array, give it dimension 4 in the subprogram's DIMENSION statement, and write:

```
COMMON X, DA, Y
```

where DA is the name of the fake array. This will not work, because of the separate handling of arrays and single variables. The effect of such a statement would be to make X and A correspond, and Y and B correspond.

It is allowable for a variable to appear both in a COMMON statement and in an EQUIVALENCE statement. When a variable not appearing in a COMMON statement is made equivalent (by an EQUIVALENCE statement) to a variable which does appear in a COMMON statement, then both of the variables will be assigned to the COMMON area; no other action would make any sense. Doing this, however, may change the order in which variables are assigned to the COMMON area, according to the following rule. When COMMON variables also appear in EQUIVALENCE statements, the ordinary sequence of COMMON variables is changed and priority is given to those variables in EQUIVALENCE statements, in the order in which they appear in EQUIVALENCE statements. For example, the combination:

```
COMMON A, B, C, D
EQUIVALENCE (C, X), (Y, B)
```

will cause variables to be assigned to the COMMON area as follows:

1st: C and X

2nd: B and Y

3rd: A

4th: D

One of the most frequent uses for the COMMON statement is in supplying "implicit" arguments to FUNCTION or SUBROUTINE subprograms. Instead of setting up a SUBROUTINE subprogram, for instance, to require arguments stating all the input and output variables, the variables can be mentioned in COMMON statements in both the calling program and the subprogram. Then no arguments at all need be written in either the SUBROUTINE or the CALL statement. With the FUNCTION statement, it is necessary to have at least one argument.

For example, the COMMON statement may be used with the SUBROUTINE subprogram to segment a program systematically and completely, in the following manner. After writing the complete program, group the statements into reasonable segments according to their usage in the program. That is, find sets of statements which generally operate together. Write a SUBROUTINE statement at the beginning of each segment (except the first, which becomes the "main" program), an END statement at the end of each segment, and enough RETURN statements to allow for proper exit from each subprogram. Write a COMMON statement containing the names of all variables, both single and arrays, which appear anywhere in any part of the program; insert this statement in the main program and every subprogram (possibly punching the cards once and reproducing them, to save writing). Similarly, make up a DIMENSION statement containing the name of every array in the entire program, and insert it into the main

program and every subprogram. The programs may now be batch compiled. Later, if it is necessary to recompile one subprogram or the main program, it may be done independently (unless, of course, something in the COMMON or DIMENSION statements is to be changed).

This approach will result in putting some things in COMMON which do not actually need to be there, but this is immaterial. The method outlined here is somewhat of a "brute force" approach.

Use of COMMON statements in connection with the partitioning of a program is illustrated in the complete example shown next in Section VIII.

SECTION VIII

SAMPLE ALGEBRAIC COMPILER PROGRAM

General Description

Suppose that for given values of x and n, it is desired to compute the quantities p1, p2, p3, p4 and p5 defined below.

$$p1 = .1\ (1-x)^{n+1}(1-(1-y)^{n}(1+ny)),$$

$$p2 = (n+1)(1-x)^{n}(1-(1-y)^{n})x+p1,$$

$$p3 = (1(1-x)^{n+1}(1-y)^{n+1})-p1,$$

$$p4 = (1-(1-x)^{n+1}-(n+1)(1-x)^{n}x+(1-x)^{n+1}(1-(1-y)^{n+1}))-p1,$$

$$p5 = (n+1)(1-x)^{n}(1-y)^{n}x,$$

where $y=x^2$.

The program written for this problem is partitioned into several subprograms to facilitate checkout. As one may readily see, the main program TABLE simply loops through the series of calls to the subprograms. The subroutines TABIN, TABCMP, TABOUT, and TABEX are the input, computation, output, and exit routines, respectively. Communication of data between subprograms is achieved via the variables in COMMON. The program goes to the exit routine when an end of file card (FINIS punched in columns 2-6) has been detected by the input routine. By utilizing sense switches, the program will permit both reading and printing to be either on-line or off-line.

The plan of execution is very simple--read a card, compute the results, write them out, and repeat the cycle until the "FINIS" card. A data card is punched with ten values of x and three values of n, to be stored into the arrays P and N respectively. In TABCMP, for each value of x and n, the quantities $p_i(i=1,5)$ are evaluated and stored as elements in corresponding arrays. When this process is completed for each set of input, the output routine simply writes out the required information in an intelligible fashion.

ALGEBRAIC COMPILER STATEMENT

TITLE _____ WRITTEN BY _____ CHECKED BY _____ DATE _____ PAGE ____ OF ____

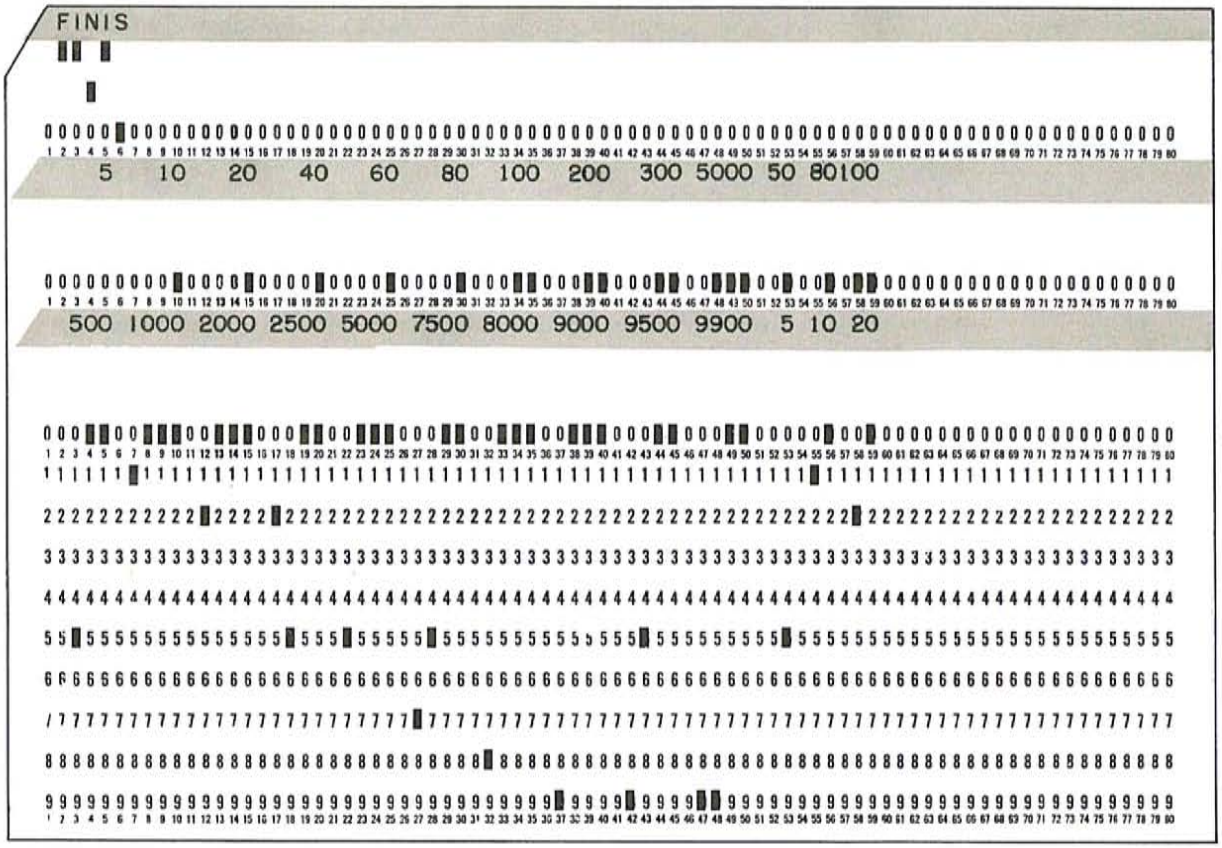
ABC	STATEMENT NUMBER	CONTIN	ALGEBRAIC COMPILER STATEMENT (1 / 11 / 23 / 38 / 52 / 66 / 72 / 80)
1			TITLE TABLE
2			COMMON P,N,P1,P2,P5,P3,P4
3			DIMENSION P(10),N(3),P1(3,10),P2(3,10),P3(3,10),P4(3,10),P5(3,10)
4	C		

```
C     MAIN PROGRAM -- CALLS ON THREE SUBPROGRAMS:
C     TABIN - INPUT,
C     TABCMP- COMPUTATION,
C     TABOUT- OUTPUT.
C          PROGRAM STOPS WHEN TABIN READS END-OF-FILE CARD AND
C     GOES TO EXIT ROUTINE.
C
      1    CALL TABIN
           CALL TABCMP
           CALL TABOUT
           GO TO 1
           END

           SUBROUTINE TABIN                                          TABCMP
           COMMON P,N,P1,P2,P5,P3,P4
           DIMENSION P(10),N(3),P1(3,10),P2(3,10),P3(3,10),P4(3,10),P5(3,10)
C
C     INPUT ROUTINE
C
           IF(SENSE SWITCH 2)10,12
     10    READ INPUT TAPE 2,40,P,N
           IF END OF FILE 50,30
     12    READ 40,P,N
           IF END OF FILE 50,30
     30    RETURN
     40    FORMAT (10F5.4,3I3)
     50    CALL TABEX                                                EXIT
           END

           SUBROUTINE TABCMP
           COMMON P,N,P1,P2,P5,P3,P4
           DIMENSION P(10),N(3),P1(3,10),P2(3,10),P3(3,10),P4(3,10),P5(3,10)
C
C     COMPUTATIONAL ROUTINE FOR THE PROBABILITY FUNCTIONS
C     P1,P2,P3,P4,P5, AT P(I) FOR A GIVEN N.
C
           DO 100 K=1,3
           DO 100 I=1,10
           Q=P(I)**2
             D1=(1.0-P(I))**(N(K)+1)
             D2=N(K)+1
             D3=(1.0-Q)**N(K)
             D4=D3*(1.0-0)
           P1(K,I)=.1*D1*(1.0-D4-D2*Q*D3)*10.E5
             D5=(1.0-P(I))**N(K)
           P2(K,I)=D2*P(I)*D5*(1.0-D3)*10.E5+P1(K,I)
           P5(K,I)=D2*P(I)*D5*D3*10.E5
           P3(K,I)=(1.0-D1*D4)*10.E5-P1(K,I)
           P4(K,I)=(1.0-D1-D2*P(I)*D5+D1*(1.0-D4))*10.E5-P1(K,I)
     100   CONTINUE                                                  TABCMP
           RETURN
           END
```

```
      SUBROUTINE TABOUT
      COMMON P,N,P1,,P2, P5, P3, P4
      DIMENSION P(10),N(3),P1(3,10),P2(3,10),P3(3,10),P4(3,10),P5(3,10)
C
C     OUTPUT ROUTINE
C
      IF(SENSE SWITCH 1)2,4
    2 WRITE OUTPUT TAPE 4,10
      GO TO 15
    4 PRINT 10
   10 FORMAT (1H1/5B,5HTABLE////14HB,
     X26HALL OUTPUTS ARE 1,000,000X///36B,4HP   6B,6H  P3   6B,6H  P4
     Y6B,6H  P1  6B,6H  P2  6B,4H  P5///)
C
   15 DO 50 K=1,3
      IF(SENSE SWITCH 1)32,34
   32 WRITE OUTPUT TAPE 4,40,N(K),((P(I),P3(K,I),P4(K,I),P1(K,I),P2(K,I),
     XP5(K,I),I=1,10)
      GO TO 50
   34 PRINT 40,N(K),((P(I),P3(K,I),P4(K,I),P1(K,I),P2(K,I),P5(K,I),I=1,10
     Y))
   40 FORMAT (1H+20B,3HN =I3/10(28B,F12.5,5F12.3/)/1)
   50 CONTINUE
      RETURN
      END
      SUBROUTINE TABEX                                          EXIT
      COMMON P,N,P1,,P2, P5,P3, P4                              ROUTINE
      DIMENSION P(10),N(3),P1(3,10),P2(3,10),P3(3,10),P4(3,10),P5(3,10)
      IF(SENSE SWITCH 1)10,20
   10 WRITE OUTPUT TAPE 4,30
      END FILE 4
      STOP 777
   20 PRINT 30
      STOP 777
   30 FORMAT (1H5 5B,13HEND OF TABLES)
C     DUMMY RETURN
      RETURN
      END
```

Sample Input Data

The following three cards represent two sets of sample input data in the form required by the above program, followed by a FINIS card.



SECTION IX

SUMMARY OF HONEYWELL ALGEBRAIC COMPILER SYSTEM

General Properties of a Source Program

An Algebraic Compiler source program consists of a sequence of source statements, of which there are 42 different types. Each statement of a source program is punched beginning on a separate card, with up to nine continuation cards allowed for statements which are too long to fit on one card. The sequence of source program statements is determined only by the sequence of cards in the source program deck. In particular, the sequence is not affected by the use of statement numbers or numbers in the continuation column (see below).

Cards containing a "C" in column 1 are not processed by the Compiler and may contain any desired comments which will appear in a listing of the source program deck. Cards containing a "B" in column 1 are treated by the Compiler as Boolean statement cards, as described below. Cards containing an "A" in column 1 are treated as ARGUS statement cards, as described below.

Any number less then 32,768 may be punched anywhere in columns 1 through 5 of the first card of a statement, which then becomes the statement number. Statement numbers of Boolean and ARGUS cards must be four or fewer digits in length, and may be punched anywhere in columns 2 through 5. Blanks anywhere in a statement number are ignored; leading zeros in a statement number are also ignored. Statement numbers may be written in any sequence, but may not be duplicated within one program. It is not necessary for all statements to have statement numbers.

Column 6 of the first card of a statement must be left blank or punched with zero. Continuation cards, other than for comment cards, must have column 6 punched with some character other than zero. A comment card may not be thought of as being continuable; every comment card must have a "C" punched in column 1. ARGUS statements are not continuable by the nature of the statements. Continuation cards for Boolean statements must have both the "B" in column 1 and the non-zero punch in column 6.

The statements themselves are punched in columns 7 through 72, both on the first and continuation cards. A table of the admissible Honeywell 800 characters and their use in Algebraic Compiler statements appears in Appendix A. Except for the "blank" and

"Hollerith" field specifications in a FORMAT statement, blank columns in statement cards are ignored.

CONSTANTS, VARIABLES, ARRAYS, AND ARITHMETIC STATEMENTS

Fixed-Point Constants

A fixed-point constant is written as one to five decimal digits. It is characterized as fixed point by being written without a decimal point and without an "E" to indicate an exponent, and is thus restricted to integer values and zero. A preceding plus or minus sign[1] is optional; if no sign is written, the number is assumed to be positive. A fixed-point constant must always be less in absolute value than $2^{15} = 32,768$.

Floating-Point Constants

A floating-point constant is written as not more than 16 characters, including sign, decimal point, and the letter E. It is characterized by having a decimal point, or the letter E to indicate an exponent, or both. The decimal point, if used, may appear at the beginning, at the end, or between any two digits. A preceding sign[1] is optional; if the number is written without a sign, it is assumed to be positive. A floating-point constant may optionally be written with a decimal exponent to indicate the power of 10 by which the number is multiplied; this is done by writing an "E" after the number, followed by the exponent. A negative exponent is indicated by a - sign; a positive exponent may be written with or without a + sign. The value of the floating-point constant must either lie between the approximate limits of 10^{-77} and 10^{+76} (2^{-256} and 2^{+252}) in absolute value, or be zero.

Variables and the Names of Variables

Four kinds of variables are permitted: fixed point, floating point, alphanumeric, and Boolean. Fixed-point variables can only be integers and are named in a different way than the others. Floating-point variables can only be values expressible as normalized floating-point numbers. Alphanumeric variables may be composed of any Honeywell 800 characters, including letters and special characters. Boolean variables, in the broadest sense, include all of the other types; the term is used primarily to describe variables appearing in Boolean statements.

Since in most problems fixed- and floating-point variables will be used much more heavily than alphanumeric and Boolean variables, and since there is no question of ambiguity involved, only two types of names are provided for the four types of variables. As prescribed below, the name of a fixed-point variable distinguishes it from the other three; the rules for

[1] Although the effect of a negative constant is easily achieved by prefixing the constant with a minus sign, negative constants as such are not generated by the Compiler.

naming floating-point variables also cover alphanumeric and Boolean variables.

In order to avoid ambiguity in the naming of variables and functions, it is necessary to observe the following two rules.

Rule 1. A variable must not be given a name which is the same as the name of a function without the final F.

Rule 2. The name of a subscripted variable must not end in F unless it is less than four characters long.

Fixed-Point Variables

The name of a fixed-point variable is one to six letters or digits, of which the first is I, J, K, L, M, or N. Punctuation marks or other special characters may not be used, and the two rules above must be observed. A fixed-point variable may take on any positive or negative integral value which is less in absolute value than 2^{44}, which is approximately 10^{13}. If it is to be used as a subscript, or as an indexing parameter in a DO statement, it must be less in absolute value than $2^{15} = 32,768$. If a fixed-point variable larger than 2^{15} is used in these latter cases, it will be reduced modulo 2^{15}, i. e., only the rightmost 15 bits will be used.

Floating-Point Variables

The name of a floating-point variable is one to six letters or digits, of which the first is alphabetic but not I, J, K, L, M, or N. Punctuation marks or other special characters may not be used, and the two rules stated under "Variables and the Names of Variables" must be observed. A floating-point variable may take on any value which is expressible as a normalized floating-point number, i. e., its absolute value must lie between the approximate limits of 10^{-77} and 10^{+76} (2^{-256} and 2^{+252}) or be zero.

Alphanumeric Variables

Alphanumeric variables are named in the same way as floating-point variables. An alphanumeric variable (note: not the name) consists of eight characters, and there is no restriction on the use of the characters. An alphanumeric variable can be entered into the computer as input, using the "A" field specification in a FORMAT statement; an alphanumeric variable may also be defined as an ARGUS constant, using the ALF pseudo operation. These are the only ways to define alphanumeric quantities; there is no way to write a combination of arbitrary characters and have it regarded as a constant, i. e., a literal value,

as with fixed- and floating-point constants. Alphanumeric variables are ordinarily used only in ARGUS statements, as arguments in CALL statements, in IF statements, in a list, in Boolean statements, and in statements of the form a = b.

Boolean Variables

Boolean variables are named in the same way as floating-point variables. A Boolean variable consists of 48 binary digits, which are often written for convenience as 16 octal digits. A Boolean variable may be entered into the computer as input, using the "O" field specification in a FORMAT statement; a Boolean variable may also be defined as an ARGUS constant using the OCT pseudo operation. However, Boolean variables are not restricted to quantities so entered. In fact, any variable may be regarded as a Boolean variable if properly handled. Boolean variables are used only in Boolean statements. There is no way to write an octal number and have it regarded as a Boolean constant, i. e., as an octal number instead of a fixed-point number.

Subscripted Variables

A subscripted variable has the name of a fixed- or floating-point variable, followed by parentheses enclosing one, two, or three subscripts separated by commas, the variable then represents an element of a one-, two-, or three-dimensional array. A subscript in a subscripted variable is a fixed-point quantity, the value of which determines the element in the array to which reference is made. A subscript may be an expression in any of the following five forms:

1. A fixed-point constant;
2. A fixed-point variable;
3. A fixed-point variable plus or minus a fixed-point constant;
4. A fixed-point constant times a fixed-point variable;
5. A fixed-point constant times a fixed-point variable, plus or minus a fixed-point constant.

A variable in a subscript must not itself be subscripted. A variable which appears in subscripted form must appear in a DIMENSION statement somewhere in the program. The value of a subscript must be greater than zero and not greater than the corresponding maximum size given in the DIMENSION statement. A subscripted variable must always be written with the same number of subscripts as appear in its DIMENSION statement.

An array is stored with the element corresponding to the subscript (1), (1, 1), or (1, 1, 1) in the lowest-numbered location and the others in consecutive ascending locations.

Two- and three-dimensional arrays are stored in consecutive locations in such a way that their first subscript (from the left) varies most rapidly and their last subscript varies least rapidly.

Expressions

An expression is a sequence of constants, subscripted or non-subscripted variables, and functions, separated by operation symbols, commas, and parentheses, and obeying the rules given below. Several of these rules have to do with the mode of an expression; every expression is of the fixed-point or floating-point mode, depending on whether the value of the expression is a fixed-point or a floating-point number. Boolean expressions are not considered in these rules (see below for a discussion of Boolean expressions). The rules are stated in such a way that all expressions may be derived from combinations of constants, variables, and functions.

Rule 1. Any fixed-point or floating-point constant, variable, or subscripted variable, is itself considered to be an expression.

Rule 2. In forming an expression, fixed-point and floating-point quantities can be mixed only in the following two ways:

a. A floating-point quantity can appear in a fixed-point expression only as an argument of a function;

b. A fixed-point quantity can appear in a floating-point expression only as an argument of a function, or as an exponent, or as a subscript.

Rule 3. A function is an expression, if expressions of the correct modes are written as its arguments. The mode of the function considered as an expression is the same as the mode of the value determined by the function.

Rule 4. If E is an expression, and if its first character is not + or -, then +E and -E are expressions of the same mode as E.

Rule 5. If E is an expression, then (E) is an expression of the same mode as E.

Rule 6. If E and F are expressions of the same mode, and if the first character of F is not + or -, then the following are all expressions of the

same mode as E and F:

E + F

E - F

E * F

E / F

The characters +, -, *, and / are used to denote addition, subtraction, multiplication, and division, respectively.

Rule 7. If E and F are expressions, and F is a floating-point expression only if E is, and if the first character of F is not + or -, and neither E nor F is of the form A ** B, then E ** F is an expression of the same mode as E. The character combination ** is used to denote exponentiation.

Hierarchy of Operations

Three rules govern the order in which operations are carried out:

Rule 1. Where parentheses are used, they override the following two rules.

Rule 2. If the hierarchy of operations is not explicitly specified by parentheses, it is taken in the following order, from innermost to outermost:

Exponentiation

Multiplication and division

Addition and subtraction

Stated otherwise, in the absence of parentheses all exponentiations are carried out first, then all multiplications and divisions, and finally all additions and subtractions.

Rule 3. Expressions in which parentheses are omitted from a sequence of consecutive multiplications and divisions, or a sequence of consecutive additions and subtractions, are treated as though there were parentheses grouped from the left. Thus, if zero represents either * or /, or separately, + or -, then A0B0C0D will be taken to mean (((A0B)0C)0D).

Arithmetic Statements

An arithmetic statement is of the general form a = b, where a is a subscripted or non-subscripted variable, and b is an expression as defined previously. The = sign is not used here in the sense of "is equivalent to", but rather is used to mean "the value defined by the expression b replaces the previous value of a".

The result of a calculation defined by an arithmetic formula is in floating-point form if the variable on the left side of the = sign has the name of a floating-point variable, and in fixed-point form if the variable on the left has the name of a fixed-point variable. If the variable on the left is fixed point and the expression on the right is floating point, the result is first computed with floating-point arithmetic, then truncated and converted to a fixed-point integer. (Truncate, as used here, means to discard any fractional part of the result without rounding). If the variable on the left is floating point and the expression on the right is fixed point, the result is computed using fixed-point arithemtic, then converted to floating-point form.

Boolean Statements

Any statement card which has "B" punched in column 1 will result in the compilation of instructions in the object program to do Boolean Algebra. This may apply to arithmetic expressions, to expressions in IF statements, in function definitions, and in the arguments of a CALL statement. In every case, Boolean algebra will be performed on all variables; Boolean and "ordinary" algebra cannot be mixed. The elements of which Boolean algebra is to be performed must have the names of floating-point variables; these may have been defined as ARGUS constants or entered with an "O" field definition in a FORMAT statement. The three allowable operations and the symbols used for them are:

1. Logical addition (inclusive OR) +
2. Logical multiplication (AND) *
3. Complementation -

The exclusive OR function of two variables may be obtained by making them the arguments of the function EXCLORF. The hierarchy of operations in a Boolean expression is as in ordinary algebra, except that there is no subtraction, division, or exponentiation; i. e., multiplication is done before addition. Complementation is a unary operation, and enough parentheses must be used to completely define which one expression is to be complemented.

ARGUS Statements

It is possible to intersperse Honeywell 800 instructions, written in the ARGUS language, into an Algebraic Compiler program. The ARGUS instructions may be written on the Compiler coding form as desired in the compiled program, by placing an "A" in column 1 of the statement line of the Compiler coding form.

The rest of the ARGUS instruction format is:

Columns 2-5: Statement Number or Blank

Columns 11-22: Operation Code

Columns 24-37: A Address

Columns 38-51: B Address

Columns 52-65: C Address

The discussion which follows assumes a rudimentary knowledge of the ARGUS system.

The allowable types of addresses used in ARGUS statements are limited to names of floating-point variables, ARGUS constants, literal floating-point constants without a sign, statement numbers or binary counts according to the following table:

Type of Operation	A Address	B Address	C Address
Arithmetic (floating binary)	General	General	Variable
Logical	Symbol	Symbol	Variable
Comparison	General	General	Statement Number
TS	General or Inactive	Variable or Inactive	Statement Number
TX	General		Variable
Shift	Symbol	Binary Count	Variable
Print	General	Inactive	Statement Number or Inactive

In the above table the following definitions apply:

Variable - name of a floating-point variable.

General - includes name of floating-point variable and ARGUS constant, and literal floating-point constants.

Symbol - Includes name of floating-point variable and ARGUS constant.

The portions of the ARGUS vocabulary which may be used include the three ARGUS constant pseudo instructions ALF, OCT, and FLBIN, with the restriction that these must appear with only one entry per statement line. In addition to these data entry instructions, the Algebraic Compiler permits the use of: BA, BS, BM, BD, WA, WD, HA, SM, SS, EX, TX, TS, NN, NA, LN, LA, SPS, SPE, SWS, SWE, PRA, PRD, PRO, FBA, FBS, FBM, FBD, FLN, FFN, and FNN.

All other ARGUS instructions are specifically excluded from the set of permissible instructions in the Algebraic Compiler. Further, ARGUS instructions may not use the cosequence mode, simulator instructions, or masking. No Compiler functions may be addressed.

CONTROL STATEMENTS

Unconditional GO TO Statement — GO TO n

Control is transferred to the statement with the statement number n.

Computed GO TO Statement — GO TO $(n_1, n_2, \ldots, n_m)$, i

In this statement, i must be a non-subscripted fixed-point variable and $n_1, n_2, \ldots, n_m$ must be statement numbers. If the value of the variable i at the time this statement is executed is j, control is transferred to the statement with the statement number n_j. The value of i must never be outside the range 1 to m.

Assigned GO TO Statement — GO TO n, $(n_1, n_2, \ldots, n_m)$

In this statement, n must be a non-subscripted fixed-point variable which appears in a previously-executed ASSIGN statement and $n_1, n_2, \ldots, n_m$ must be statement numbers. Control is transferred to the statement having for its statement number whichever one of the values $n_1, n_2, \ldots, n_m$ was most recently assigned to n by an ASSIGN statement.

ASSIGN Statement — ASSIGN n_i TO n

In this statement, n must be a non-subscripted fixed-point variable which appears in an assigned GO TO statement, and n_i must be one of the statement numbers appearing in parentheses in the same assigned GO TO. When the assigned GO TO is next executed, control is transferred to the statement with the statement number n_i, unless another applicable ASSIGN intervenes.

CONTINUE Statement — CONTINUE

This is a dummy statement which does not result in any instructions in the object program. It is used primarily as the last statement in the range of a DO, when needed to satisfy the requirement that the range of a DO must not end with any statement which can cause a transfer of control.

IF Statement — IF (e) n_1, n_2, n_3

In this statement, e is an expression and n_1, n_2, n_3 are statement numbers. Control is transferred to the statement with the statement number n_1, n_2, or n_3, depending on whether the value of e is less than zero, equal to zero, or greater than zero, respectively.

IF PARITY Statement — IF PARITY n_1, n_2

This statement may be used to alter the course of a computation upon detection of an uncorrectable parity error on a magnetic tape. In the statement, n_1 and n_2 must be statement numbers. If there was a parity error on the preceding statement and the parity routines were not able to correct it, the statement number n_1 is executed next; if there was no error or if it was corrected, the statement with the statement number n_2 is executed next. The IF PARITY statement may optionally follow any of the statements READ TAPE, WRITE TAPE, READ INPUT TAPE, or WRITE OUTPUT TAPE; if so, it must be the next executable statement. If the statements IF PARITY and IF END OF FILE are both used, the IF PARITY should be first. If an uncorrectable error is detected and there is no IF PARITY statement following the input or output statement, the object program will print an error indication and stop.

IF END OF FILE Statement — IF END OF FILE n_1, n_2

This statement may be used to alter the course of a computation under any of the following conditions:

1. In connection with a READ TAPE statement, upon detection of the indication written on a magnetic tape by the END FILE statement;
2. In connection with a READ, READONE, or READTWO statement, upon detection of a card with the word FINIS punched in columns 2 through 6;
3. In connection with a READ INPUT TAPE statement, upon detection of a record produced by a card with the word FINIS punched in columns 2 through 6.

In the statement, n_1 and n_2 must be statement numbers. If the relevant condition was detected in connection with the preceding input or output statement, the statement with the statement number n_1 is executed next; if the condition was not detected, the statement with the statement number n_2 is executed next. The IF END OF FILE statement must be the next executable statement after the input or output statement to which it refers, except that an IF PARITY statement may intervene. If any of the conditions listed above are detected and there is no IF END OF FILE statement following the input or output statement, the object program will produce an error indication and stop.

DO Statement — DO n i = n_1, n_2, n_3

or DO n i = n_1, n_2

In this statement, n must be a statement number, i must be a non-subscripted fixed-point variable, and n_1, n_2, and n_3 must each be either an unsigned fixed-point constant or

a non-subscripted fixed-point variable. If n_3 is not written, as in the second form of the statement, it is assumed to be 1.

The statements following the DO, up to and including the statement with the statement number n, are executed repeatedly. They are executed first with $i = n_1$; on each following execution i is increased by n_3. Repeated execution is continued until the statements have been executed with i equal to the largest value which does not exceed n_2.

The range of a DO is the set of repeatedly executed statements. Stated otherwise, it is the set of statements beginning with the statement immediately following the DO and continuing up to and including the statement with the statement number n.

The index of a DO is the fixed-point variable i. Throughout the execution of the range, i is available for use in computation, either as a fixed-point variable or as the variable of a subscript. The value of i is also available for use in computation if control passes to statements outside of the range. This may happen either by the execution of control statements which cause a transfer of control outside of the range of the DO, or by the normal completion of the number of executions of the range as specified by n_1, n_2, and n_3. In the latter case, the DO is said to be satisfied. The following rules must be observed in writing DO statements:

Rule 1. If the range of one DO (the "outer" DO) contains statements in the range of another DO (the "inner" DO), then all the statements in the range of the inner DO must also be in the range of the outer DO. (This does not prohibit having the ranges of two or more DO's end with the same statement.)

Rule 2. The last statement in the range of a DO must not be a statement that can cause a transfer of control.

Rule 3. No statement may be executed within the range of a DO, which redefines or otherwise alters the value of the index of the DO or of n_1, n_2, or n_3.

Rule 4. Control must not transfer into the range of a DO from a statement outside its range, with one exception: it is permissible to transfer control out of the range of a DO, perform a series of calculations, and then transfer control back to the same section of the range of the DO from which exit was made. When this is done, the statements to which control is transferred are called the extended range of the DO. It is still necessary to observe Rule 3 in the

extended range. Furthermore, if there are any DO's in the extended range, the transfer is only permitted from the innermost DO of a completely-nested set of DO's, i.e., every pair of DO's in the "nest" is such that one contains the other.

PAUSE Statement — PAUSE or PAUSE n

If the second form of the statement is used, n must be an unsigned octal constant which may contain as many as five digits. When the statement is encountered in the object program, the following are typed out on a console typewriter:

1. The title of the main program, which appeared in the TITLE statement;
2. The word PAUSE;
3. The constant n, or nothing if the first form of the statement is used;
4. The status of the simulated sense lights and sense switches.

The machine then waits for the operator to take some action. If a restart at the sequence register is initiated, the program will continue execution, beginning with the next executable statement after the PAUSE. It is possible to change the status of the simulated sense switches before continuing. Any programs being parallel processed will not be affected by these actions.

STOP Statement — STOP or STOP n

If the second form of the statement is used, n must be an unsigned octal constant which may contain as many as five digits. When the statement is encountered in the object program, the following are typed on a console typewriter:

1. The title of the main program, which appeared in the TITLE statement;
2. The word STOP;
3. The constant n, or nothing if the first form of the statement is used;
4. The status of the simulated sense lights.

The execution of this program is then halted and control is returned to the Executive Routine. Manually starting this program again at the sequence register will have unpredicted results, since it is not anticipated that the programmer will want to continue execution of a program after encountering a STOP. The execution of this statement will not affect any programs being parallel processed with this one.

SENSE LIGHT Statement[1] — SENSE LIGHT i

This statement provides a means of indicating conditions in a problem both to the operator and to other portions of the program. The value of i must lie in the range of zero through 4. If i is zero, all sense lights (1 through 4) will be turned off, i.e., SENSE LIGHT 0 in effect clears all sense lights. If i has any other value, i.e., 1 through 4, that particular sense light

[1] The SENSE BIT Statement may be used interchangeably with the SENSE LIGHT Statement.

will be turned on. For example, SENSE LIGHT 3 turns on sense light 3. For a discussion of sense lights and sense switches, see Appendix B.

IF (SENSE LIGHT) Statement[1] IF (SENSE LIGHT i) n_1, n_2

This statement is used to alter conditionally the sequence of the execution of statements dependent upon the status of one of the sense lights. In the statement, n_1 and n_2 are statement numbers and i is the number of a sense light, 1 through 4. If sense light i is in the on condition, control is transferred to statement number n_1, otherwise control is transferred to statement number n_2. If the sense light is on at the time of execution of this statement, it will be turned off. In other words, sense light i is always left in the off condition as the result of the execution of this statement.

IF (SENSE SWITCH) Statement[2] IF (SENSE SWITCH i) n_1, n_2

This statement is similar to the IF SENSE LIGHT statement except that the sense switches (see Appendix B) are interrogated rather than the sense lights. n_1 and n_2 are statement numbers and i identifies the sense switch used. The value of i may range from 1 through 6. Control is transferred to statement n_1 if sense switch i is down and to statement n_2 if sense switch i is up.

IF ACCUMULATOR OVERFLOW Statement IF ACCUMULATOR OVERFLOW n_1, n_2

This statement is used to control the program sequence depending on the setting of a switch which is set by an addition or subtraction overflow unprogrammed transfer. Control is transferred to statement number n_1 if an accumulator overflow has occurred or to statement number n_2 if overflow has not occurred since the previous IF ACCUMULATOR OVERFLOW statement. The use of this statement resets the internal indicator tested.

IF QUOTIENT OVERFLOW Statement IF QUOTIENT OVERFLOW n_1, n_2

This statement is used to test the status of a switch set by an exponential overflow or underflow unprogrammed transfer. Control is transferred to statement n_1 if an exponent has been created by any of the floating-point operations that is greater than +63 or less than -64, or to statement n_2 if these exponent limits have not been exceeded since the previous IF QUOTIENT OVERFLOW statement. The use of this statement resets the internal indicator tested.

---

[1] The IF (SENSE BIT) Statement may be used interchangeably with the IF (SENSE LIGHT) Statement.

[2] The IF (SENSE FLAG) Statement may be used interchangeably with the IF (SENSE SWITCH) Statement.

IF DIVIDE CHECK Statement — IF DIVIDE CHECK n_1, n_2

This statement is used to test a switch set by a division overcapacity unprogrammed transfer. Control is transferred to statement n_1 if a division instruction has been attempted that cannot be performed, or to statement n_2 if no illegal divisions have been attempted since the previous IF DIVIDE CHECK statement. The use of this statement resets the internal indicator tested.

TITLE Statement — TITLE Name

This statement must be on the first card of every program. The word TITLE must be punched in columns 2 through 6 of the statement card, and the desired title in columns 7 through 14. The name used should not duplicate any already on the collector tape. If no TITLE statement is provided, a dummy name will be supplied by the Compiler. A TITLE statement is not required by FUNCTION and SUBROUTINE subprograms, and is ignored if present; the name of the subprogram becomes its name on the collector tape. A TITLE card must not be preceded by a blank card. A TITLE may not begin with an asterisk as it might otherwise be confused with principle subroutines.

END Statement — END

This is a non-executable statement which must be on the last card of every program or subprogram deck. It is required in order to separate programs in batch program compilation, but it is nevertheless required even though only one program is being compiled. The word END must be punched in columns 7 through 9 of the statement; any other punches on the card are ignored. An END card must not be followed by a blank card.

FINIS Statement — FINIS

This is a non-executable statement which must be on the last card of the batch of program decks being compiled together. It is required even if the batch consists of only one program. No other characters may be written in this statement, besides the word FINIS, which must appear in columns 2 through 6 of the statement card.

INPUT AND OUTPUT STATEMENTS

Definition of a List

All of the input and output statements which transmit information require a list in order to specify the variables to be transferred between storage and an input or output device, and to specify the sequence in which they are to be transferred. A list consists of the names of the variables to be transferred, together with parenthesized indexing information to specify how the subscripted variables (if any) are to be treated. The variables

are transferred in the order in which they are named, from left to right, with repetition of any variables for which indexing information is supplied. Parenthesized variables with indexing information may be thought of as being equivalent to implicit DO loops, where each opening parenthesis (except subscripting parentheses) corresponds to a DO, with its indexing information written just before the matching closing parenthesis, and with its range ending with the indexing information. As in a DO statement, the indexing information consists of three fixed-point constants or fixed-point variables; if the last of these is omitted, it is assumed to be 1. "Nests" of indexing information may be be at most three levels deep. If the list of an input statement is written in the form I, A(I), or the form J, (A(I), I = 1, J), or in any other form in which a subscript or an indexing parameter itself appears earlier in the list, the subscripting or indexing will be carried out using the new value.

If it is desired to transfer an entire array, the name of which must of course appear in a DIMENSION statement, it is permissible to omit the indexing information. When this is done, the elements of the array are understood to be ordered in the same way that the elements of an array are ordered in storage, i. e., with the first subscript from the left varying most rapidly and the last subscript varying least rapidly.

FORMAT Statement — FORMAT (Field Specifications)

All of the input and output statements which require a list, with the exception of READ TAPE and WRITE TAPE, require, in addition, the statement number of a FORMAT statement which describes the information format to be used. The FORMAT statement also describes, in some cases, the kind of conversion to be performed between the internal and external representation of the information to be transferred. A FORMAT statement is not executable, and may be placed anywhere in the source program.

In the discussions below, the term "unit record" is used. Depending on which input or output statement is used, a unit record may consist of:

1. A line to be printed on an on-line printer, with a maximum of 120 characters;
2. A punched card to be read from an on-line card reader or punched on a directly-connected punch, with a maximum of 80 characters;
3. An alphabetic tape record to be read or written, with a maximum of 120 characters;
4. A binary tape record to be read or written, with any number of words. The number of physical tape blocks in such a record is determined by considerations in the object program and is of no concern to the programmer.

The field specification in a FORMAT statement describes the unit record(s) involved by giving, for each field in the record, from left to right beginning with the first character of the record:

1. The type of information and/or the type of conversion to be used; this is done with the seven field specification characters discussed below;
2. The number of characters in the field;
3. For some of the field specifications certain other information is required and/or may optionally be given; these cases are discussed in connection with the description of the field specifications below.

If a number of consecutive fields, say n, have the same format and type of conversion, n may be written before the field-specification character to so indicate. This repetition of groups of field specifications may be called for by enclosing the group of field specifications within parentheses and writing the desired number of repetitions in front of the opening parenthesis. Only one level of grouping is permitted, i. e., parentheses within parentheses are not permitted for this purpose.

When the list of an input or output statement is used to transfer more than one unit record, with the different records having different formats, a slash is used to separate the format specification of each record. It is possible to specify that the first one or more records have a special format, and that all following records have the same format; this is done by enclosing the last record specification in a second set of parentheses. A slash always indicates the end of one record and the beginning of a new one; the closing parenthesis of the FORMAT statement always indicates the end of a record. The skipping of entire records, which in practice usually means the printing of blank lines, is called for by writing consecutive slashes. The skipping of n records is called for by writing n + 1 slashes.

With the exceptions of a FORMAT statement which consists entirely of Hollerith fields and of the "B" (blank) field specification (see below), a FORMAT statement is always used in conjunction with the list of an input or output statement. The list specifies the variables to be transferred and in what order, and the associated FORMAT statement, the format of each variable as well as the length of each record if there is more than one. As the object program transmits the variables named in the list, it scans the FORMAT statement, from left to right, to find the proper field specification for each variable; any repetition of field specification or of groups of field specifications is, of course, taken into account. Whenever Hollerith field specifications (see below) are encountered in scanning the FORMAT statement, they are dealt with in the proper place, without any transmission of variables from the list. The transmission of variables is terminated only when all items in the list have been transmitted, but any

remaining Hollerith field specifications in the FORMAT statement will be dealt with even after the transmission of the last variable. If the last field specification in the FORMAT statement has been used and items named in the list remain to be transmitted, scanning of the FORMAT statement begins again with the first field specification in the last set of parentheses in the statement.

Scale Factor

A scale factor is optional with the "E" and "F" field specifications. It is written as n P s where n is the scale factor, s is the field specification and P is used only as a separation character. Its usage is different for the two types; details are discussed below. Four conventions which apply to both types may be mentioned here:

1. Once a scale factor has been given, it applies to all "E" and "F" field specifications in the same FORMAT statement, until another scale factor appears in the scanning of the FORMAT statement;
2. If no scale factor is given, it is taken to be zero. Once a scale factor has been given, a scale factor of zero must be given in order to return to the normal mode;
3. Scale factors apply only to the "E" and "F" field specifications, and with the "E" type only to output. Use of a scale factor with any other type of field specification has no effect; use of a scale factor for input of "E" fields produces unpredicted results and no error indication is given;
4. When a scale factor is written with a field specification which includes a repetition number, the repetition number is written between the scale factor and the E or F. If there is no repetition number given, i. e., if it is understood as 1, then it may be written or not. Thus, in the case of the "F" field specification, for instance (see below), the following are all permissible: 3P4F12.4, 3PF12.4, 3P1F12.4; the last two are equivalent.

Field Specification "E" (Floating Point) Ew.d

The "E" field specification is used to indicate conversion between an internal floating-point variable and an external floating-point decimal number, i. e., one written with an explicit exponent. The total number of characters in the field, including sign, decimal point, exponent, and any blanks, is specified by w. The number of decimal places after the decimal point is specified by d; d is treated modulo 10.

On input, a sign, if it appears, must be the first non-blank character of the field. The use of a + is always optional. The use of a decimal point is optional; if it is not supplied, then the position of the assumed decimal point is given by d, but if it is supplied its position overrides d. Blanks embedded in the number are to be zeros. The "number" part of the field must

not exceed 14 digits, not counting sign, blanks, or decimal point. The exponent part of the field is of the general form E±ee, where ee is the numeric exponent, but several simplifications are permitted. A positive exponent may appear with the + omitted or replaced by a blank, i. e., in the forms E ee or Eee. If the most significant digit of the exponent is zero, it may be omitted. If the exponent appears with a sign, the E may be omitted. Thus, the following are all permissible (and equivalent) forms for the exponent plus two: E + 02, E 02 E02, E + 2, E 2, E2, +02, +2. A scale factor has no effect on input with the "E" field specification.

On output, the number will appear at the right of the field, if w is larger than the number of characters in the field. If w is not large enough to contain the converted internal number, leading characters will be lost. There will be no embedded blanks in the field, with the exception that + signs are not entered but are replaced by blanks. In the absence of a scale factor (see below), the field will appear in the form ±0.nn.....E±ee (except that any + signs do not actually appear) where the number of places after the decimal point is specified by d.

A positive scale factor may be used by writing the field specification in the form sPnEw.d, where s is the scale factor, and n is the number of repetitions for the field specification (see the discussion of scale factor conventions under the general discussion of the FORMAT Statement). The effect of the use of a scale factor in this case is to move the decimal point s places to the right and to decrease the exponent by s.

Field Specification "F" (External Fixed Point) Fw.d

The "F" field specification is used to indicate conversion between an internal floating-point variable and an external fixed-point number, i. e., one written without an exponent. The total number of characters in the field, including sign, decimal point, and any blanks, is specified by w. The number of decimal places after the decimal point is specified by d; d is treated modulo 10.

On input, a sign, if it appears, must be the first non-blank character of the field. The use of a + is always optional. The use of a decimal point is optional; if it is not supplied, then the position of the assumed decimal point is given by d, but if it is supplied its position overrides d. Blanks embedded in the number are to be zeros. The number must not exceed 12 digits, not counting sign, blanks, or decimal point.

A positive or negative scale factor may be used for input with the "F" field specification; the effect is to multiply the external number by 10 to the negative of the scale factor. If the scale factor is s, the formula is:

Internal number = External number · 10^{-s}

On output, the number will appear at the right of the field, if w is larger than the number of characters in the field. If w is not large enough to contain the converted internal number, leading characters will be lost. There will be no embedded blanks in the field. Positive numbers appear without a + sign.

A positive or negative scale factor may be used, by writing the field specification in the form ±sPnFw.d, where the + sign is optional, s is the scale factor, and n is the number of repetitions of the field specification (see the discussion of scale factor conventions under the general discussion of the FORMAT Statement). The effect of the use of the scale factor in this case is to move the decimal point of the external number s places to the right if s is positive, or to the left if s is negative. Stated otherwise, s is a number such that:

External number = Internal number · 10^s

Field Specification "I" (Integer) Iw

The "I" field specification is used to indicate conversion between an internal fixed-point variable and an external decimal integer. The total number of characters in the field, including sign and any blanks, is w.

On input, a sign is optional; if it appears, it must be the first non-blank character in the field. The use of a + sign is always optional. The use of a decimal point is, of course, not permitted. Blanks embedded in the number are assumed to be zeros. The number must not exceed 14 digits, not counting sign or blanks; for use as a subscript or index the limit is in fact 32,767.

On output, the number will appear at the right of the field, if w is larger than the number of characters in the field. If w is not large enough to contain the converted internal number, leading characters will be lost. Positive integers appear without the + sign.

Field Specification "H" (Hollerith) wH

The Hollerith field specification does not call for the output of a variable, but the output of the following text itself. The w characters immediately following the letter H, where w may be any integer not exceeding the size of the record, are placed in the record in the position indicated by the position of the Hollerith field specification in the FORMAT statement. Any Honeywell 800 character may be used, including the character blank; this is the only instance in which a blank in a statement is not ignored. The characters printed by the high- and standard-speed printers available with the Honeywell 800 are different in a few cases; reference should be made to the character configuration table in Appendix A to determine

what characters will be printed.

Indication of the presence of Hollerith text is not required in the list of the output statement which refers to the FORMAT statement. Whenever a Hollerith field specification is encountered in the scanning of the FORMAT statement, the following text is written out and scanning continues without any variables having been transmitted. It is possible to write a line consisting entirely of Hollerith text, by using an output statement with no list, and which refers to a FORMAT statement with only the "H" field specification. The Hollerith text is not available to the programmer for use in any way other than for input or output.

When a FORMAT statement containing Hollerith text is referenced by an input statement, the listed text is replaced by whatever text appears in the corresponding field of the input record. When the same FORMAT statement is later used with an output statement, the text which has been "read into" the FORMAT statement will then be transferred to the output record. The text thus entered is still not available to the programmer for use in any other way than for input or output. (The "A" field specification, described below, is available for use in entering alphabetic data, which can then be manipulated by the program.)

For all output statements that result in printing, e.g., PRINT and WRITE OUTPUT TAPE, single spacing of the printed lines will result unless specific control is given otherwise. This is accomplished through the use of the Hollerith field specification in a FORMAT statement. If a field specification 1H is used as the first field specification in a FORMAT statement associated with an output statement, no data per se is transmitted, but rather the one Hollerith character is interpreted as a control character. The permissible characters and their interpretation are:

Blank - single space after the current line is printed
+ - suppress spacing after the current line is printed
0 - double space after the current line is printed
1 - space to head of form after the current line is printed
2-9 - this number of lines are to be spaced after the current line is printed

If any other character is used in this connection, it will be placed in the output area. If this Hollerith field specification is used in connection with punching, this control character will not be punched.

Field Specification "O" (Octal) Ow

The "O" field specification is used to indicate conversion between an internal 48-bit Honeywell 800 word and an external fixed-point octal integer. The total number of characters in the field, including sign and any blanks, is w, which must be 16 or less.

On input, if 8, 9 or any other illegal character appears in the field, it will be converted to alphanumeric form and the right three bits stored; no error indication will be given. Blanks embedded in the field are assumed to be zeros.

For signed fields, the sign must be the first non-blank character of the field. If w is 16, the leftmost digit must not exceed 3. If w is less than 16, the converted number will be placed in the right side of the storage location. If the sign is +, all four sign bits of the Honeywell 800 word will be set to 1; if the sign is -, all four sign bits will be set to zero.

If the field appears without a sign, the characters of the field are stored in the left side of the storage location, with no special handling of the sign bits of the Honeywell 800 word. The leading digit may be any octal digit.

On output, the number will appear at the right of the field, if w is larger than the number of characters in the field. If the field defined by w is not large enough to contain the converted internal number, leading characters will be lost. Positive integers appear without a + sign.

Field Specification "A" (Alphabetic) Aw

The "A" field specification is used to indicate conversion between an internal 48-bit Honeywell 800 word, considered as eight alphanumeric characters, and an external field consisting of any combination of Honeywell 800 characters. The number of characters in the field, w, may not exceed eight.

On input, if w is less than eight, the field will be stored in left-justified form, i. e., the first character of the field will appear in the leftmost character of the computer word, and the extra characters at the right end of the computer word will be filled with blanks.

On output, if w is less than eight, the w characters at the left end of the computer word will be moved to the output record.

Field Specification "B" (Blank)[1] wB

On input, the "B" field specification calls for the next w character positions in the input record to be skipped over. No indication in the list of the input statement referencing the FORMAT statement is required.

On output, the "B" field specification calls for w blanks to be inserted into the output

[1] The field specification "X" may be used interchangeably with specification "B".

record. No indication in the list of the output statement referencing the FORMAT statement is required.

READ Statement — READ n, List

In this statement, n is the statement number of a FORMAT statement, and the list is as described above. The READ statement calls for the reading of cards from the on-line card reader designated as number 1. As many cards are read as are required to supply the number of variables specified in the list. The arrangement of information on the cards is defined in the FORMAT statement; each field is converted, also as defined in the FORMAT statement, and placed in the computer storage location assigned to the corresponding variable named in the list.

The detection of a card with the word FINIS punched in columns 2 through 6 constitutes an end-of-file condition for this statement. If the statement is executed when there are no cards in the card reader ready to be read, the program will wait for cards to be made ready.

READ ONE Statement — READ ONE n, List

This statement is exactly equivalent to the READ statement.

READ TWO Statement — READ TWO n, List

This statement is equivalent to the READ and READ ONE statements, except that cards are read from the on-line card reader designated as number 2.

PRINT Statement — PRINT n, List

In this statement, n is the statement number of a FORMAT statement, and the list is as described above. The PRINT statement calls for lines to be printed on the on-line printer designated as number 1. As many lines are printed as are required to exhaust the list. The arrangement of information in the lines is defined in the FORMAT statement; each variable in the list is converted, also as defined in the FORMAT statement, and printed, one line at a time.

PRINT ONE Statement — PRINT ONE n, List

This statement is exactly equivalent to the PRINT statement.

PRINT TWO Statement — PRINT TWO n, List

This statement is equivalent to the PRINT and PRINT ONE statements, except that lines are printed on the on-line printer designated as number 2.

PUNCH Statement — PUNCH n, List

In this statement, n is the statement number of a FORMAT statement, and the list is as described above. The PUNCH statement calls for Hollerith cards to be punched on the on-line punch designated as number 1. As many cards are punched as are required to exhaust the list. The arrangement of information on the cards is defined in the FORMAT statement; each variable in the list is converted, also as defined in the FORMAT statement, and punched, one card at a time.

PUNCH ONE Statement — PUNCH ONE n, List

This statement is exactly equivalent to the PUNCH statement.

PUNCH TWO Statement — PUNCH TWO n, List

This statement is equivalent to the PUNCH and PUNCH ONE statements, except that cards are punched on the on-line punch designated as number 2.

READ INPUT TAPE Statement — READ INPUT TAPE i, n, List

This statement is used to read a magnetic tape which contains up to 80 Honeywell 800 characters in alphanumeric form. In the statement, i is an unsigned fixed-point constant in the range of zero through 63, and must be the number of a magnetic tape unit which is available on the computer system to be used by the object program. Symbolic tape addresses are not permitted. The statement number of a FORMAT statement is given by n, and the list is as described previously. As many records are read as are required to exhaust the list; the length of each record is determined by the FORMAT statement, which also determines the type of conversion applied to each variable in the list.

Detection of the record produced by a card with the word FINIS punched in columns 2 through 6 constitutes an end-of-file condition for this statement, along with detection of the physical end of tape.

WRITE OUTPUT TAPE Statement — WRITE OUTPUT TAPE i, n, List

This statement is used to write a magnetic tape containing records of up to 120 Honeywell 800 characters in alphanumeric form. In the statement, i is an unsigned fixed-point constant in the range of zero through 63, and must be the number of a magnetic tape unit which is available on the computer system to be used by the object program. Symbolic tape addresses are not permitted. The statement number of a FORMAT statement is given by n, and the list is as described previously. As many records are written as are required to exhaust the list; the length of each record is determined by the FORMAT statement, which

also determines the type of conversion applied to each variable in the list. An END FILE statement should be given after writing the last record.

READ TAPE Statement — READ TAPE i, List

This statement is used to read magnetic tapes produced by a WRITE TAPE statement, without any type of conversion; note that no FORMAT statement is referenced by a READ TAPE statement. In the statement, i is an unsigned fixed-point constant in the range of zero through 63, and must be the number of a magnetic tape unit which is available on the computer system to be used by the object program. Symbolic tape addresses are not permitted. The list is as described previously. As many records are read as are required to exhaust the list. A READ TAPE statement can read any tape produced by a WRITE TAPE statement.

End-of-file conditions are initiated by detection of the indication written by an END FILE statement.

WRITE TAPE Statement — WRITE TAPE i, List

This statement is used to write a magnetic tape which contains Honeywell 800 words exactly as they appear in storage, without any type of conversion; note that no FORMAT statement is referenced by the WRITE TAPE statement. In the statement, i is an unsigned fixed-point constant in the range of zero through 63, and must be the number of a magnetic tape unit which is available on the computer system to be used by the object program. Symbolic tape addresses are not permitted. The list is as described previously.

END FILE Statement — END FILE i

This statement is used to write on a magnetic tape a signal which can be recognized by the IF END OF FILE statement for binary tapes and by the off-line printer for alphanumeric tapes. In the statement, i is an unsigned fixed-point constant in the range of zero through 63, and must be the number of a magnetic tape unit which is available on the computer system to be used by the object program.

REWIND Statement — REWIND i

This statement is used to rewind a magnetic tape to the beginning of the tape. In the statement, i is an unsigned fixed-point constant in the range of zero through 63, and must be the number of a magnetic tape unit which is available on the computer system to be used by the object program.

BACKSPACE Statement — BACKSPACE i

This statement is used to backspace a magnetic tape by one record. It applies equally to tapes produced by the WRITE TAPE and WRITE OUTPUT TAPE statements and to tapes prepared by an off-line card reader. In the statement, i is an unsigned fixed-point constant in the range of zero through 63, and must be the number of a magnetic tape unit available on the computer system to be used by the object program.

BUFFER Statement — BUFFER $(n_1, n_2, n_3), (m_1, m_2, m_3)\ldots$

The BUFFER statement makes it possible to overlap reading and computation, writing and computation, or reading, writing, and computation, in the case of the READ TAPE and WRITE TAPE statements. The list of such a statement must consist of the name of exactly one array, shown in non-subscripted form. Each buffer assigned must be large enough to hold the largest array which it will have to handle. All buffer areas may be set up with one BUFFER statement, if desired, or a separate statement may be used for each. Use of input buffering requires an additional special register group, as does output buffering.

The number of additional memory words used by buffering is, approximately:
Reading: 150 + sum of lengths of input buffers + 105
Writing: 150 + length of longest buffer + 105

The symbols used in the specimen statement above are to be interpreted as follows:
n_1 = IN for reading, OUT for writing
n_2 = number of the tape unit involved
n_3 = number of words in the longest record to be read or written with this tape

ERASE Statement — ERASE (List)

This executable statement clears to zero the locations corresponding to the variables specified in the list.

FUNCTIONS

General Considerations

Some of the material below requires a basic understanding of the concept of the Collector. The collector is a magnetic tape on which are "collected" all the compiled programs which are available for operation on the computer at an installation. The output of a compilation is a set of records added to the collector tape, along with certain optional listings. The output

of a compilation, however, is not in the final form on the collector tape, but rather is on the tape in sections, ready to be "collected" together to form a running program. This final collection is done prior to run time; it may be done immediately at the completion of compilation or at any later time. Certain control cards may optionally be used to alter the collection process; these are discussed in connection with FUNCTION and SUBROUTINE subprograms.

Open Functions

An open function is one which is compiled into the object program each time it is used at the point where it is brought into operation in the source program. It is the only one of the five types of functions with this characteristic. The Honeywell Algebraic Compiler, as supplied, contains 13 such functions.

The name of an open function consists of four to seven alphabetic or numeric characters (but no special characters), of which the first must be alphabetic and the last F. The first character must be X if, and only if, the value of the function is to be fixed point. The name of the function is followed by parentheses enclosing the argument(s), which are separated by commas if there is more than one. Each open function has a prescribed mode (fixed or floating point) for its argument(s) and for its value; different functions must be used for each combination of modes of argument(s) and function value. The output of an open function always consists of one value. Any expression, including another function, may be used as an argument of an open function.

The 13 open functions which are supplied with the Honeywell Algebraic Compiler are shown in Figure 6.

Library Functions

A library function is compiled into the object program only once, regardless of how many times it is used. The Honeywell Algebraic Compiler, as supplied, contains 15 such functions.

Any appearance of the name of a library function causes the compilation of the function into the object program. Every appearance of the name brings the function program into operation.

The name of a library function consists of four to seven alphabetic or numeric characters (but no special characters), of which the first must be alphabetic and the last F. The first character is X if, and only if, the value of the function is fixed-point. The name of the function is followed by parentheses enclosing the argument(s), which are separated by commas if there is more than one. Each library function has a prescribed mode (fixed or floating point) for its argument(s) and for its value; different functions must be used for each combination of modes of argument(s) and function value. The output of a library function always consists of one value. Any expression, including another function, may be used as an argument of a library function.

The 15 library functions which are supplied with the Honeywell Algebraic Compiler are shown in Figure 7.

Defined Functions

A defined function (which is also called an arithmetic statement function) is one which is defined with a single statement and then brought into operation elsewhere in the source program wherever its name appears. The definition and use of defined functions are at the discretion of the source programmer and are independent of any open or library functions. A defined function applies only to the program or subprogram in which it appears.

A defined function is defined to the Compiler by a statement of the form a = b, where a is the function name and b is an expression. The name of a defined function is four to seven alphabetic or numeric characters (but no special characters), of which the first must be alphabetic and the last F. The first character must be X if, and only if, the value of the function is to be fixed point. The name of the function is followed by parentheses enclosing the argument(s), which are separated by commas if there is more than one. In the definition statement, the arguments must be distinct non-subscripted variables. The right-hand side of the definition statement may be any expression which does not involve subscripted variables. The right-hand side may involve variables not specified as arguments, and may make free use of other functions.

The variables appearing as arguments in the definition of a defined function are only dummies which, in effect, specify to the Compiler how to substitute into the defined function

the arguments which are written when the defined function is later used. Therefore, the variable names used in the function definition are unimportant, except as they indicate fixed- or floating-point variables, and may be the same as the names of actual variables appearing elsewhere in the program.

The program which is compiled to carry out the operations specified in the function definition statement appears once in the object program. Each time the defined function is used, the object program then refers to the one place where the defined function appears. A defined function is thus compiled as a closed subroutine.

A defined function may be used anywhere in a program, by writing the name of the function and writing for arguments any expressions which agree in number, order, and mode, with the arguments as stated in the definition of the function. In particular, the arguments may be subscripted variables, constants, or functions. The output of a defined function always consists of one value.

FUNCTION Subprograms

A FUNCTION subprogram is one which is defined by the use of a FUNCTION statement followed by any number of Algebraic Compiler statements, and then brought into operation elsewhere in the program by writing the name of the function. A FUNCTION subprogram is an independent part of a total program. Its variable names may be the same as names which appear in the main program or in other subprograms or it may have its own DIMENSION and EQUIVALENCE statements; any defined functions appearing in a FUNCTION subprogram apply only to that subprogram. The arguments in a FUNCTION statement may be the names of arrays as well as the names of single variables. The output of a FUNCTION subprogram always consists of one value. A FUNCTION subprogram may be compiled with a main program and/or other subprograms, or it may be compiled independently.

FUNCTION Statement FUNCTION Name $(a_1, a_2, \ldots, a_n)$

The name of a FUNCTION subprogram consists of one to six alphabetic or numeric characters, the first of which must be alphabetic; the first character must be I, J, K, L, M, or N if, and only if, the value of the function is to be fixed point, and the last character must not be F if the name is more than three characters in length. The name must not appear in a DIMENSION statement in the FUNCTION subprogram, nor in a DIMENSION statement in any program which uses the subprogram. The name of the function must appear at least once in the FUNCTION subprogram as a variable on the left-hand side of an arithmetic statement, or, alternatively, in an input statement list. The name of the

function is followed by parentheses enclosing the argument(s), which are separated by commas if there is more than one. In the FUNCTION statement, the arguments must be distinct non-subscripted variables appearing on the right-hand sides of executable statements of the subprogram. There may be any number of arguments as long as there is at least one.

The variables appearing as arguments in the FUNCTION statement are only dummies which, in effect, specify to the Compiler how to substitute into the subprogram the arguments which are written when the subprogram is used elsewhere in the program. Therefore, the variable and array names used as arguments are unimportant, except as specifying fixed- or floating-point variables, and may be the same as names appearing in the main program or in other subprograms. However, none of the dummy variables of a FUNCTION subprogram may appear in EQUIVALENCE or COMMON statements in the subprogram. The dummy arguments must not be subscripted.

The FUNCTION statement must be the first statement of the subprogram. An END statement must be the last (physically) statement of the subprogram. The appearance of the FUNCTION statement indicates that all up to the END statements are the subprogram. The FUNCTION subprogram may use any type of statement, except other FUNCTION statements or SUBROUTINE statements. If a COMMON statement is used in the subprogram, it of course refers to the one common storage area which is the same for all programs which are collected together. This provides a means of establishing correspondence between variables in the subprogram and variables in the main program or in other subprograms, a correspondence which does not exist otherwise (even between variables having the same name). The dummy variables which are used as arguments in the FUNCTION subprogram are in non-subscripted form, but there is no such restriction on variables not used as dummy variables. Free use may be made of all types of expressions, including all of the five types of functions. A FUNCTION subprogram must contain at least one RETURN statement which must be the last statement in the sequence of execution.

The object program which is compiled to carry out the operations specified in a FUNCTION subprogram will appear in the object program once, regardless of how many times the subprogram is used. Each time the subprogram is used, the object program refers to the one place where the subprogram appears in storage; it is thus compiled as a closed subroutine.

A FUNCTION subprogram must not be written between two statements of another program. A FUNCTION subprogram may be compiled independently or batch compiled

with a main program and/or other subprograms.

A FUNCTION subprogram may be brought into operation by writing the name with arguments which agree in number, order, and mode with those in the FUNCTION statement. Furthermore, when a dummy argument is the name of an array, the corresponding real argument must also be an array name. The dummy array name must appear in a DIMENSION statement in the subprogram, and the actual array name must appear in a DIMENSION statement in the calling program, and both must have the same dimensions. Dummy variables which represent single variables may be replaced with any expressions, including subscripted variables, constants, other functions, etc.

SUBROUTINE Subprograms

A SUBROUTINE subprogram is one which is defined by the use of a SUBROUTINE statement followed by any number of Algebraic Compiler statements, and then called into operation elsewhere in the program by the use of the CALL statement. A SUBROUTINE subprogram is an independent part of a total program. Its variable names may be the same as names which appear in the main program or in other subprograms; it may have its own DIMENSION and EQUIVALENCE statements; any defined functions appearing in a SUBROUTINE subprogram apply only to that subprogram. The arguments in a SUBROUTINE statement may be the names of arrays as well as the names of single variables. The arguments may represent input to the SUBROUTINE subprogram or output from it, and the output may consist of any number of values, including arrays. A SUBROUTINE subprogram may be batch compiled with a main program and/or other subprograms, or it may be compiled independently.

SUBROUTINE Statement — SUBROUTINE Name $(a_1, a_2, \ldots, a_n)$

The name of a SUBROUTINE subprogram consists of one to six alphabetic or numeric characters (but no special characters), the first of which must be alphabetic and the last must not be F if the name is more than three characters in length. The name must not appear in a DIMENSION statement in the SUBROUTINE subprogram, nor in a DIMENSION statement in any program which uses the subprogram. The name of the subprogram is followed by parentheses enclosing the argument(s), which are separated by commas if there is more than one. In the SUBROUTINE statement, the arguments must be distinct non-subscripted variables appearing in executable statements in the subprogram; input arguments must appear in the right-hand sides of statements, and output arguments in the left-hand sides of statements. There may be any number of arguments, or none. If there are no arguments, no parentheses are required.

The variables appearing as arguments in the SUBROUTINE statement are only dummies which, in effect, specify to the Compiler how to substitute into the subprogram the arguments which are written in CALL statement(s) when the subprogram is used elsewhere in the program. Therefore, the variable and array names used as arguments are unimportant, except as specifying fixed- or floating-point variables, and may be the same as names appearing in the main program or in other subprograms. However, none of the dummy variables of a SUBROUTINE subprogram may appear in EQUIVALENCE or COMMON statements in the subprogram.

The SUBROUTINE statement must be the first statement of the subprogram. An END statement must be the last statement physically of the subprogram. The appearance of the SUBROUTINE statement indicates that all subsequent statements, up to the END statement, are in the subprogram.

The SUBROUTINE subprogram may use any type of statement except FUNCTION or another SUBROUTINE statement. If a COMMON statement is used in the subprogram, it refers to the one common storage which is the same for all programs which are collected to be run together. This provides a means of establishing correspondence between variables in the subprogram and variables in the main program or in other subprograms, a correspondence which does not exist otherwise (even between variables having the same name). The dummy variables which are used in the SUBROUTINE statement must appear in the SUBROUTINE subprogram in non-subscripted form, but there is no such restriction on variables not used as dummy variables. Free use may be made of expressions, including all of the five types of functions. A SUBROUTINE subprogram must contain at least one RETURN statement which must be the last executed statement.

The object program which is compiled to carry out the operations specified in a SUBROUTINE subprogram will appear once in the object program, regardless of how many times the subprogram is used. Each time the subprogram is used (by calling it with a CALL statement), the object program refers to the one place where the subprogram appears in storage; it is thus compiled as a closed subroutine. A SUBROUTINE subprogram must not be written between two statements of another program. A SUBROUTINE subprogram may be batch compiled with a main program and/or other subprograms, or it may be compiled independently.

CALL Statement — CALL Name (a_1, a_2, ..., a_n)

This statement is used to call into operation the SUBROUTINE subprogram specified

by the name in the CALL statement. Control is transferred to the named subprogram, and the parenthesized arguments replace the corresponding dummy arguments of the SUBROUTINE statement. The arguments in the CALL statement must agree in number, order, and mode with those in the SUBROUTINE statement. Furthermore, when an argument in the SUBROUTINE statement is an array name, the corresponding argument in the CALL statement must also be an array name. The array name in the SUBROUTINE statement must appear in a DIMENSION statement in the subprogram, and the array name in the CALL statement must appear in a DIMENSION statement in the calling program, and the dimensions must be the same. Dummy variables which represent single variables may be replaced with any expressions, including subscripted or non-subscripted variables, constants, other functions, etc., but not with literal alphabetic or numeric characters.

If a CALL statement refers to a SUBROUTINE which has been designated for overlaying (see below) and the routine is not in memory when the CALL is executed, then the object program automatically brings the subprogram into memory and then transfers control to it.

RETURN Statement — RETURN

This statement terminates any FUNCTION or SUBROUTINE subprogram, and returns control to the calling program. A RETURN statement must therefore be the last-executed statement of a subprogram. It need not, however, be physically the last statement of a subprogram; it may appear at any point in a subprogram, and there may be any number of RETURN statements in a subprogram.

Overlaying may be used with SUBROUTINE subprograms only. All subprograms to occupy the same area of memory must be named on OVERLAY control cards at collection time. When this is done, the operation of the CALL statement is modified so that if the subprogram is not in memory when called, it will be brought it.

The NEGLECT control card may be used at collection time when parts of a program have not yet been completed and it is desired to test other parts. Any FUNCTION or SUBROUTINE subprograms which have not been completed, and which are mentioned in the parts which are to be tested, should be named on NEGLECT control cards.

SPECIFICATION STATEMENTS

DIMENSION Statement — DIMENSION v, v, v, ...

The DIMENSION statement is used to specify the maximum sizes of arrays; every

variable in a program which appears in subscripted form must appear in a DIMENSION statement. In the general form of the statement shown above, v is the name of a variable with one, two, or three subscripts (unsigned fixed-point constants) in parentheses. The number of subscripts determines the number of dimensions of the array; the value of each subscript in the DIMENSION statement determines the maximum value of the corresponding subscript anywhere else in the program. The appearance of a variable in a DIMENSION statement causes space to be reserved for the array, and indirectly causes the assignment of a specific storage location for each element of the array.

Any number of subscripted variables separated by commas may appear in one DIMENSION statement, and there may be any number of DIMENSION statements in a program. A DIMENSION statement applies only to the main program or subprogram in which it appears. A DIMENSION statement may appear anywhere in a program. A DIMENSION statement must not include the name of the program in which it appears, nor the name of any FUNCTION or SUBROUTINE subprogram which the program uses.

EQUIVALENCE Statement — EQUIVALENCE (a, b, c, . . .), (d, e, f, . . .), . . .

The EQUIVALENCE statement makes it possible to assign two or more variables to the same storage locations, where the logic of the problem permits it, thus making possible significant reductions in storage space. In an alternative interpretation, the statement may be used to establish two different symbols as standing for the same variable.

The variables within a set of parentheses, which may be subscripted with a single unsigned fixed-point constant, are assigned to the same location. There may be any number of variables within one set of parentheses, any number of parentheses, and any number of EQUIVALENCE statements. Quantities which are not mentioned in EQUIVALENCE statements are assigned to unique locations. Locations can only be shared among variables, not constants.

The meaning of a subscript of a variable in an EQUIVALENCE statement is defined as follows. For a subscript greater than zero, the meaning of C(p) is: The (p - 1)th location after the one containing C, or if C is an array, the (p - 1)th location after the one containing C(1), C(1, 1), or C(1, 1, 1). Subscripting of variables in an EQUIVALENCE statement cannot be used to change the standard way in which arrays are stored. It must be emphasized that a variable may have only one subscript in an EQUIVALENCE statement, regardless of

how many dimensions it has. For two- and three-dimensional arrays, it is necessary to take into account the manner in which arrays are stored in order to compute the subscript required in an EQUIVALENCE statement to establish equivalence between some single variable(s) and a specific element of an array. If it is desired to establish an equivalence involving the first element of an array, the array name may be written in either of the forms A or A(1). In order to establish equivalence between the first locations of a number of arrays, which need not have the same dimensions or total number of locations, it is satisfactory to write the names of the arrays in non-subscripted form.

An EQUIVALENCE statement applies only to the program or subprogram in which it appears.

COMMON Statement — COMMON A, B, C, . . .

The COMMON statement makes it possible to establish correspondence between variables in different subprograms. Variables which are not mentioned in a COMMON statement are assigned to locations in the same general section of storage as the instructions of the program in which they appear; variables named in a COMMON statement are assigned to a special COMMON area which is separate from all programs. The variables named in a COMMON statement apply only to the program or subprogram in which they appear, but the COMMON area is the same for all programs and subprograms which are collected together.

In a COMMON statement, single variables and arrays are treated separately, and there is a part of the COMMON area for each. The single variables are assigned to successive locations in the single-variable part of the COMMON area, in the order in which they appear in COMMON statements, regardless of how they may be interspersed among the names of arrays. (However, this sequence may be altered if variables appear both in COMMON and EQUIVALENCE statements.) Similarly, arrays are assigned to the array part of the COMMON area, in the order in which they appear in the statement, with enough space being assigned to contain each array. This process is applied to all COMMON statements in a program. Therefore, the first single variable appearing in a COMMON statement in a program is assigned to the first single-variable location in the COMMON area. In this way, correspondence is established between variables in different subprograms, whether or not they have the same names. It would be pointless, although not damaging, to have a COMMON statement in one subprogram or in the main program, without having another COMMON statement elsewhere in the program.

When COMMON variables also appear in EQUIVALENCE statements, the ordinary sequence of COMMON variables is changed and priority is given to those variables in EQUIVALENCE statements, in the order in which they appear in EQUIVALENCE statements.

When it is necessary to put variables into corresponding positions in two COMMON statements, it is permissible to make up variable or array names which do not actually appear in the subprogram in question. Because of the separate treatment of single variables and arrays, however, made-up arrays must not be used to force correspondence between single variables.

Key Punch	Card Code	Honeywell 800 Code	Octal	Standard Printer	High Speed Printer	Console	Key Punch	Card Code	Honeywell 800 Code	Octal	Standard Printer	High Speed Printer	Console
0	0	000000	00	0 (zero)	0	0	-	X	100000	40	- (minus)	-	-
1	1	000001	01	1	1	1	J	X, 1	100001	41	J	J	J
2	2	000010	02	2	2	2	K	X, 2	100010	42	K	K	K
3	3	000011	03	3	3	3	L	X, 3	100011	43	L	L	L
4	4	000100	04	4	4	4	M	X, 4	100100	44	M	M	M
5	5	000101	05	5	5	5	N	X, 5	100101	45	N	N	N
6	6	000110	06	6	6	6	O	X, 6	100110	46	O	O	O
7	7	000111	07	7	7	7	P	X, 7	100111	47	P	P	P
8	8	001000	10	8	8	8	Q	X, 8	101000	50	Q	Q	Q
9	9	001001	11	9	9	9	R	X, 9	101001	51	R	R	R
	8, 2*	001010	12	9*	'	'		X,8,2*	101010	52	R*	#	#
#	8, 3	001011	13	=	=	=	$	X,8,3	101011	53	$	$	$
@	8, 4	001100	14	- (minus)	:	:	*	X,8,4	101100	54	*	*	*
Space	Blank	001101	15	Blank	Blank	Blank		X,8,5*	101101	55	**	"	↓
	8, 6*	001110	16	=*	Blank*	¢		X,8,6*	101110	56	$*	Blank*	?
	8, 7*	001111	17	- (minus)*	&	&		X, 0	101111	57	0*	Blank*	◆
&	R	010000	20	+	+	+		8, 5*	110000	60	-(minus)*	Blank*	
A	R, 1	010001	21	A	A	A	/	0, 1	110001	61	/	/	/
B	R, 2	010010	22	B	B	B	S	0, 2	110010	62	S	S	S
C	R, 3	010011	23	C	C	C	T	0, 3	110011	63	T	T	T
D	R, 4	010100	24	D	D	D	U	0, 4	110100	64	U	U	U
E	R, 5	010101	25	E	E	E	V	0, 5	110101	65	V	V	V
F	R, 6	010110	26	F	F	F	W	0, 6	110110	66	W	W	W
G	R, 7	010111	27	G	G	G	X	0, 7	110111	67	X	X	X
H	R, 8	011000	30	H	H	H	Y	0, 8	111000	70	Y	Y	Y
I	R, 9	011001	31	I	I	I	Z	0, 9	111001	71	Z	Z	Z
	R, 8, 2*	011010	32	I*	;	;		0,8,2*	111010	72	Z*	@	@
	R, 8, 3	011011	33	.	.	.	,	0,8,3	111011	73	,	,	,
□	R, 8, 4	011100	34	)	)	)	%	0,8,4	111100	74	(	(	(
	R, 8, 5*	011101	35	)*	%	%		0,8,5*	111101	75	(*	C_R	C_R
	R, 8, 6*	011110	36	.*	■	■		0,8,6*	111110	76	,*	Blank*	□
	R, 0	011111	37	0*	Blank*	△		0,8,7*	111111	77	(*	Blank*	⊗

Notes: Key Punch: Use MLT PCH key to overpunch omitted characters.

Card Code: * legal in illegal punch check Mode 2 only for card readers.

Printer: * indicates symbol which will be printed by otherwise non-standard printer bit configuration.

APPENDIX B
SENSE LIGHTS AND SENSE SWITCHES

Some computing equipment is constructed with lights and switches as part of the console or control panel. The Honeywell 800, however, uses a typewriter and keyboard so that all communication between the operator and the hardware and vice versa results in printed copy for semi-permanent or permanent records. In the interest of compatibility with other computing systems, however, sense lights and sense switches are simulated by the Algebraic Compiler and hence the Compiler statements which refer to them are valid. When they occur in hardware form, both sense lights and sense switches are bi-stable devices whose state may be interrogated by the program. The major difference between them is that sense switches are manually set by the operator under instructions from the programmer, while sense lights are set and cleared by the program itself.

A sense switch may be thought of as a toggle switch something akin to the switch we use to turn on the light in our homes. As such it is either in an up or down condition and its condition may be altered only by the operator.

A sense light is a light which the program turns on and off as an indicator to itself. One use made of these lights is a test for overflow. In case of overflow, we may wish to continue our program, yet be cognizant of the fact that overflow has occurred. This may be accomplished by having a sense light turned on, once overflow occurs. At the conclusion of the program, the status of the sense light may be tested to ascertain whether or not overflow did occur.

APPENDIX C

LIMITS ON SOURCE PROGRAM IMPOSED BY TABLE SIZES

The following are maximum quantities for an entire source program unless otherwise specified:

Table	2 Bank	4 Bank
ARGUS Constants	50	150
Fixed-Point Variables	100	300
Floating-Point Variables	300	1000
Fixed-Point Constants (other than 0-7)	42	192
Floating-Point Constants (other than 0.0 and 1.0)	146	296
Variables in Common	200	750
Dimensioned Variables	100	300
Equivalenced Variables	250	750
Sets of Equivalences	125	275
Non-Executable Statements	49	199
DO Statements	65	149
Statements Controlling Flow	199	499
Statement Numbers in All Computed GO TO's	99	199
Statement Numbers in All Assign GO TO's	99	199
Assigned GO TO Statements	49	99
where: a = number of Dimensioned Variables b = number of Equivalenced, Non Dimensioned Variables		
then: $2a + b =$	250	750
Sigma Tau Table (number of unique subscript combinations used in the program)	300	900
Subprogram Dummy Variable Table Space	50	100
Algebraic Restriction (see below)		
$3L + 6m + 4n + 3p =$	250	750

The following restrictions apply to the 2 - bank system only.

if:

c = number of unique three-dimensional subscript combinations in a program

d = number of unique two-dimensional subscript combinations in a program

e = number of one-dimensional subscript combinations in a program

f = number of subscript combinations included in c, d and e above in which the first subscript in the combination is a non-complex expression of the form (i), where i is a fixed-point variable (e.g., type c: (3I, J + 2, 1) type c and f: (I, J + 2, 1))

then:

$6c + 4d + 2e - f \leq 300$

if:

g = the number of dimensioned dummy variables of a subroutine or function program

h = the number of non-dimensioned dummy variables

then:

$2g + h \leq 49$, and $h + g \leq 48$

if:

j = the number of variables in an equivalence with a position given (e.g., A(1))

k = the number of variables in the equivalence without a position given (e.g., A)

then:

$2j + k$ (for any single set) ≤ 20

Algebraic Expressions

The following are limits on any single expression found in an IF statement, as a CALL argument, as a function definition or the right-hand side of an algebraic statement:

	2 Bank	4 Bank
Let n be the number of elements and operators in an expression, then	$n \leq 80$	596
If the expression is a defined function, let f be the number of dummy variables	$f \leq 20$	100
If L is the number of left parentheses to a point and R is the number of right parentheses, then at any point	$(L - R) \leq 14$	47
Let C be the number of commas other than those appearing in a subscript, and n is defined as above, then	$C \leq 2\left[84-n\right] - 2$	$s\left[600-n\right] - 2$

then:

$31 + 6m + 4n + 3p \leq 250$

and:

$n \leq 14$ deep (this cannot exceed 47 in any size machine)

The number of dummy variables in a defined function definition is further limited by the complexity of the expression defining the funtion. Assuming no other rule is violated, if d = the number of dummy variables then $1 + 2m + n + d + p \leq 101$.

Input-Output Restrictions

	2 Bank		4 Bank
Defined Functions	50		125
Subroutine or Function References	300		600
Backspace Tape Statements	50		100
I-0 Statements	300		500
END FILE Statements	50		100
IF END OF FILE Statements	50		100
IF PARITY Statements	50		100
Buffered Writes	25		25
Buffered Reads	25		25
DO's In a Nest	50		150
Register Variables In a Nest	100	(Appearances)	250
Control Variables, Relative Constants and DO Parameters (which are Register variables) In a Nest	75	(Unique)	200
IFNTAB	1000		2292
SUFIXTAB	200		200

Supplementary Internal Table Restrictions

The limit of IFNTAB is 1000 entries.

In a main program:

let:

a = the number of DO's in the program

b = defined functions in a program

c = the number of arguments of CALL statements or of function references which are dimensioned and subscripted. Note, if the argument is of a function reference which is an argument of a CALL, it need only be counted once.

In a subprogram:

let:

d = the number of arguments of CALL statements or of function references which are arguments of the subprogram but which are not dimensioned.

e = the number of return statements in a subroutine.

then:

IFNTAB = a + b + c in a main program

IFNTAB = a + b + c + d + e in a subprogram

The limit of SUFIXTAB is 200 entries.

let:

c and d are defined as for IFNTAB above but computed for a single source statement.

then:

the maximum number of the SUFIXTAB entries required by this statement will be

1 + c in a main program

1 + c + d in a subprogram

APPENDIX D

SOURCE PROGRAM STATEMENTS AND SEQUENCING

The rules governing the sequence of execution of source program statements are as follows:

1. The first executable statement in the deck as compiled is executed first;
2. If statement S has just been executed, then the next statement executed is dictated by the normal sequencing properties of statement S, as shown in the table below. If, however, S is the last statement in the range of one or more DO's which are not yet satisfied, then DO sequencing takes precedence.

The statements FORMAT, DIMENSION, EQUIVALENCE, COMMON, TITLE, END, FINIS, and BUFFER are non-executable statements. In questions of sequencing, they may be ignored.

The last statement in every source program deck must be an END statement, and the last statement in every batch of program decks must be a FINIS statement. The statement preceding the END statement should be a STOP, RETURN, IF, or GO TO. If this requirement is not met, the Compiler will give a diagnostic error indication and the compilation will not be completed.

Every executable statement in an Algebraic Compiler source program, except the first, must have some path of control leading to it.

TABLE OF SOURCE PROGRAM STATEMENT SEQUENCING

Statement	Normal Sequencing
a = b	Next executable statement
GO TO n	Statement n
GO TO (n_1, n_2, ..., n_m), i	Statement n_i
IF (e) n_1, n_2, n_3	Statement n_1, n_2, or n_3, if (e) is < , =, or > , zero respectively.
GO TO n, (n_1, n_2, ..., n_m)	Statement number last assigned to n
ASSIGN n_i TO n	Next executable statement
IF PARITY n_1, n_2	Statement n_1 if there was an uncorrectable error on the preceding tape operation, otherwise n_2

TABLE OF SOURCE PROGRAM STATEMENT SEQUENCING (cont)

Statement	Normal Sequencing
IF END OF FILE n_1, n_2	Statement n_1 if there was an end-of-file condition encountered on the preceding input or output operation, otherwise n_2
CONTINUE	Next executable statement
DO n i = n_1, n_2 or DO n i = n_1, n_2, n_3	DO-sequencing, then next executable statement
PAUSE or PAUSE n	Next executable statement
STOP or STOP n	Terminates program execution
SENSE LIGHT i	Next executable statement
IF(SENSE LIGHT i) n_1, n_2	Statement n_1 if sense light i is on, otherwise n_2
IF(SENSE SWITCH i) n_1, n_2	Statement n_1 if sense switch i is down, otherwise n_2
IF ACCUMULATOR OVERFLOW n_1, n_2	Statement n_1 if an overflow condition is present, otherwise n_2
IF QUOTIENT OVERFLOW n_1, n_2	Statement n_1 if an overflow condition is present, otherwise n_2
IF DIVIDE CHECK n_1, n_2	Statement n_1, if the divisor is less than the dividend, otherwise n_2
TITLE	Not executed
END	This statement terminates compilation of the program
FINIS	No sequencing; the statement terminates all compilation of the batch
FORMAT (Field Specification)	Not executed
READ n, List	Next executable statement
READ ONE n, List	Next executable statement
READ TWO n, List	Next executable statement
PRINT n, List	Next executable statement
PRINT ONE n, List	Next executable statement
PRINT TWO n, List	Next executable statement
PUNCH n, List	Next executable statement

TABLE OF SOURCE PROGRAM STATEMENT SEQUENCING (cont)

Statement	Normal Sequencing
PUNCH ONE n, List	Next executable statement
PUNCH TWO n, List	Next executable statement
READ INPUT TAPE i, n, List	Next executable statement
WRITE OUTPUT TAPE i, n, List	Next executable statement
WRITE TAPE i, List	Next executable statement
READ TAPE i, List	Next executable statement
END FILE i	Next executable statement
REWIND i	Next executable statement
BACKSPACE i	Next executable statement
BUFFER (n_1, n_2, n_3), (m_1, m_2, m_3), ...	Not executed
ERASE (List)	Next executable statement
FUNCTION Name $(a_1, a_2, \ldots, a_n)$	Not executed
SUBROUTINE Name $(a_1, a_2, \ldots, a_n)$	Not executed
CALL Name $(a_1, a_2, \ldots, a_n)$	First executable statement of the named SUBROUTINE
RETURN	The statement in the main program following the statement which caused the transfer of control to the subprogram
DIMENSION v, v, v, ...	Not executed
EQUIVALENCE (a, b, c, ...), (d, e, f, ...), ...	Not executed
COMMON A, B, ...	Not executed

Honeywell

Electronic Data Processing