H 3 207

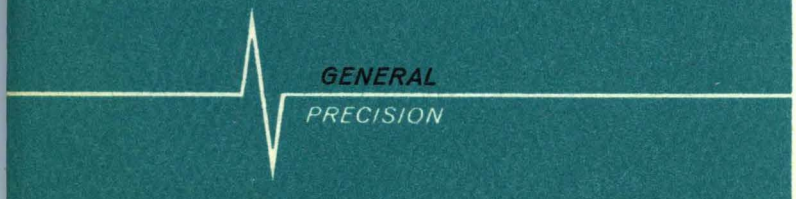
LGP 30

ROYAL PRECISION COMPUTER SYSTEM

ACT III

UNION CARBIDE CORPORATION'S
ALGEBRAIC COMPILER-TRANSLATOR

PROGRAMMER'S MANUAL



ACT III

An Algebraic Compiler for the LGP-30 Computer

PROGRAMMER'S MANUAL

By
Henry J. Bowlden
Parma Research Laboratory
Union Carbide Corporation
Parma 30, Ohio

and
Roberta R. Smith
Presently with
Stanford University

Distributed by

POOL
LGP-30, RPC-4000, and RPC-9000
Computer Users Organization
1532 N. Cahuenga Boulevard
Los Angeles 28, Calif.

TABLE OF CONTENTS

INTRODUCTION

The process of obtaining the solution to a numerical problem using ACT III is divided naturally into three portions or phases:

1. The programming phase. The first requirement is that the problem be reduced to a sequence of statements in ACT language. This source program is punched on paper tape manually, using a special typewriter called a flexowriter.

2. The compiling phase. At this point the source program is translated by the translation program (ACT IIIA) into LGP-30 machine language, and the resulting machine language program is punched into a paper tape by the computer for subsequent use. This is referred to as the object program.

3. The running phase. The object program combined with a set of service routines, referred to as the package, constitutes a working program with which data are processed and answers produced.

The present manual is designed as a guide in the first or programming phase. It contains the basic vocabulary and rules of syntax by which ACT language programs are to be constructed.

A companion manual, the Operator's Manual, describes the steps involved in carrying out the second and third phases on the computer. Many errors which consist of violations of the rules of syntax described in the present manual are detected in the compiling phase, and errors in overall logic which are detectable by incorrect answers or invalid intermediate results are detected in the running phase. The existence of these detection mechanisms is acknowledged in this volume, while details of the error displays are included in the Operator's Manual, together with a discussion of remedies.

A third manual, the Technical Manual, describes the logic of the translator itself, and also describes the symbolic assembler (SPAR) which is used to assemble packages and by which most of the language may be changed to suit varying individual needs.

An attempt has been made to organize the presentation of material in this manual in such a way that concepts may be developed at each step by means of illustrative examples. Elements of the language are introduced in each chapter, along with syntactic rules governing their usage. At the end of each chapter discussing operators, a list of the operators is given, together with the necessary information concerning precedence, operands and restrictions. This information is presented in an abbreviated form, the symbols having been explained in each chapter. These symbols are collected in a table in Appendix A. Appendix B contains a summary of the operators with reference to the text sections discussing them. The discussion of operators begins with Chapter IV and the terms used above are discussed there.

Throughout the manual, definitions will be marked by the symbol, ">". Some terms whose definitions are buried in the text of this introduction are defined here for reference.

>phase: One of the three basic segments of the process of solving a problem using the ACT III system. These are: phase 1 (programming), phase 2 (compiling) and phase 3 (running).

>compile time: A common synonym for phase 2.

>run time: A common synonym for phase 3.

>source program: The sequence of statements in ACT language, as punched manually on paper tape, required to solve a problem.

>object program: The machine language program punched by the computer in phase 2.

>package: A set of service routines in machine language used together with the object program in phase 3. The package assumed throughout this manual is labeled P-5-B.

CHAPTER I

ARITHMETIC OF THE SYSTEM

Two types of arithmetic are available in ACT III; integer arithmetic and floating-point arithmetic. Each has its advantages and its limitations, as will be discussed. Because of this dual arithmetic there are correspondingly two classes of operators, two classes of variables, and two classes of constants. Although the system allows converting a number from one type to the other, this is never done automatically. It is up to the programmer to not mix arithmetic within the same calculation.

A. Integer Arithmetic

The maximum number of significant figures allowed by the system, and the most accurate representation of a number is obtained with integer arithmetic. (For the most part integer arithmetic is exact; exceptions are noted under the specific operators.) However, the range of numbers which can be used, without causing overflow, is restricted, since 536,870,911 (positive or negative) is the largest integer which may be stored in the computer. In most cases, the bulk of the arithmetic of a problem is best handled with floating-point numbers. However, integers and integer arithmetic should be used for counters and subscripts.

B. Floating-Point Arithmetic

Floating-point arithmetic uses numbers represented in a type of scientific notation. They have the special form

$$y = a \times 10^{m},$$

where "y" is the number, "a" is a fraction (positive or negative), and "m" is an integer such that

$$0.10000002 \leq |a| \leq 0.99999994$$

and

$$-32 \leq m \leq +31 \quad .$$

In this manual then, "fraction" and "exponent" are defined as follows.

>fraction: The fraction is the fractional part of a floating-point number. e.g. "a" in the notation above.

>exponent: The exponent of a floating-point number is the power of ten by which the fraction is multiplied, e.g. "m" in the notation above. As a general rule, numbers larger in magnitude than $.99999994 \times 10^{31}$ cannot occur in a calculation without causing an error indication. Numbers smaller in magnitude than 0.1×10^{-32} are usually replaced by zero.

Of course, the advantage of floating-point arithmetic is the increase of the range of numbers which can be used with the system; however, it is not possible for it to be exact. There are two basic reasons. In the first place, the fraction must be rounded after each operation to fit the word size. In the second place, problems associated with nondecimal internal arithmetic cause small changes in the representation of even common values. For example, in floating-point, 1.0 is represented internally as 0.99999994. In general, the conversion error is no greater than 3 in the eighth significant figure. For these reasons, we do not recommend that floating-point numbers be used for counting, or that a zero result of a floating-point operation be used as a test criterion.

C. Mixed Arithmetic

As was mentioned before, mixed arithmetic is not permitted. There are operators which convert a number from its floating-point form to its integer representation, and vice versa. The compiler makes no systematic check against mixed arithmetic. At run time floating-point division with an integer denominator usually gives an "e2" stop. (See Operator's Manual.) For the most part, however, the only indication of mixed arithmetic is a wrong answer. However, if the number should be an integer and is factors of ten too large this may indicate mixed arithmetic; or if the number should be in floating-point and the fraction has leading zeros as printed out, this may also indicate mixed arithmetic. So, it is up to the programmer to keep the arithmetic of a particular calculation (or within a given "statement", Chapter III, C) consistent; and it is important that he do so.

CHAPTER II

SAMPLE PROBLEM AND PROGRAM

Although it is somewhat early, it is hoped that the introduction here of a specific problem, the mathematics of which most readers learned in high school, and the program for obtaining the answers on the computer, will serve two purposes: first, that it will help the beginner visualize correctly the general aspects of the problem-to-program process; second, that right from the beginning all readers will have in front of them a working program which demonstrates at least some syntax of the language to be discussed in the succeeding chapters.

A. Problem

The problem is to find the roots of a quadratic equation. Any quadratic equation may be reduced to the form,

$$ax^2 + bx + c = 0 \quad .$$

Then,

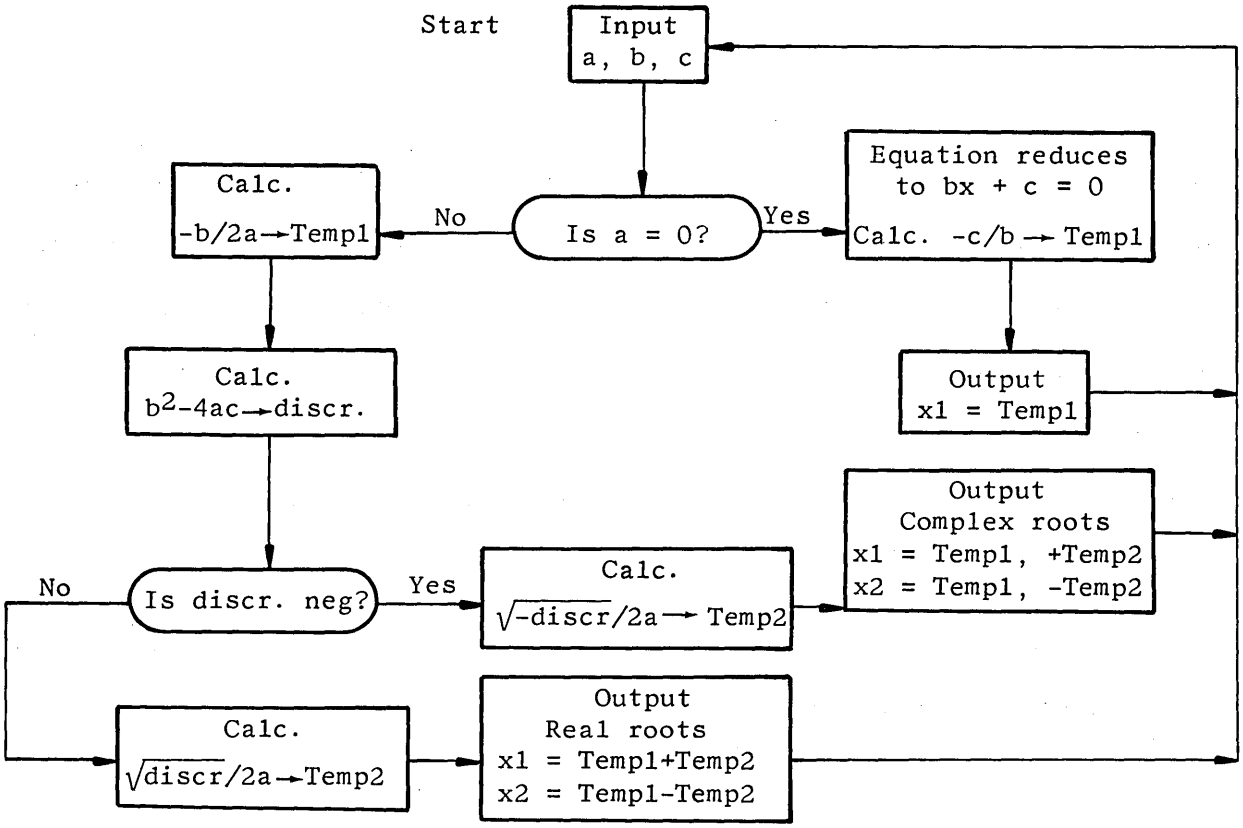
$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad .$$

If b^2-4ac is positive the roots are real and unequal.

If b^2-4ac is zero, the roots are real and equal.

If b^2-4ac is negative the roots are complex and unequal.

B. Flow of Program



C. Program

Suppose also that at run time we want the output to be self-explanatory. Therefore, in the input and output block of the program, instructions must be set up which control the flexowriter to print headings and labels in the appropriate positions. To be specific, we want the format at run time as follows:

Roots of Quadratic Equation

a = ( )* ( )**

b = ( )* ( )**

c = ( )* ( )**

Roots Real Imaginary

x1 = ( )*** ( )***

x2 = ( )*** ( )***

a = ( )* ( )**

b = ( )* ( )**

c = ( )* ( )**

Roots Real Imaginary

x1 = ( )*** ( )***

x2 = ( )*** ( )***

etc.

---

*Number entered in floating point by operator.

**Printed back in decimal by machine.

***Answers printed by machine.

For the convenience of printing back the input parameters in decimal form, as well as the roots, a, b, and c are limited to three figures following the decimal point.

A listing of the program as it is punched on a tape to be compiled by ACT III is on the left below. It finds the roots of the quadratic equation and controls the format of input and output. The numbers in the center column are for reference purposes and refer to a particular line of the program; they are not part of the program.

Program	Explanation
Problem: Roots of Quadratic Equation	1) Program labelled in a
Date: 6/7/61 Programmer: RRS'	2) remark
	3
daprt'cr4'uc2'R'lcl'o'o't's' 'o'f'	4)
'uc2'Q'lcl'u'a'd'r'a't'i'c' ''	5) Print heading
daprt'uc2'E'lcl'q'u'a't'i'o'n'.''	6)
	7
s1' daprt'cr4'cr4'cr4'a' 'uc2'='lcl' ''	8)
read'a''	9) Read "a", "b", and "c"
1603'dprt'a''	10) as floating-point
daprt'cr4'b' 'uc2'='lcl' ''	11) numbers
read'b''	12) and print them back
1603'dprt'b''	13)
daprt'cr4'c' 'uc2'='lcl' ''	14)
read'c''	15)
1603'dprt'c''	16)
daprt'cr4'cr4'uc2'R'lcl'o'o't's''	17
12'reprt' ''	18 Print
daprt'uc2'R'lcl'e'a'l''	19 labels
7'reprt' ''	20 for
daprt'uc2'I'lcl'm'a'g'i'n'a'r'y''	21 results.
daprt'cr4' 'x'l' 'uc2'='lcl''	22
	23
if'a'zero's5''	24 Is "a" zero?
.2''e'l'x'a';'2a''	25 here if "a" not zero
0-'b'/'2a';'Templ''	26 Calculate "-b/2a"
b'x'b'-'.4''e'l'x'a'x'c';'discr''	27 Calculate "discr"
if'discr'neg's2''	28 Is "discr" negative?
sqrt'discr'/'2a';'Temp2''	29 here if "discr" not neg.
1603'dprt'Templ'+'Temp2''	30 Print two
ret's6'use's7''	31 real roots
1603'dprt'Templ'-'Temp2''	32 and
use'sl''	33 return to beginning.
	34
s2' sqrt'['0-'discr']'/'2a';'Temp2''	35 Here if "discr" negative.
1603'dprt'Templ''	36
1603'dprt'Temp2''	37 Print two
ret's6'use's7''	38 complex
1603'dprt'Templ''	39 roots
1603'dprt'0-'Temp2''	40 and
use'sl''	41 return to beginning.
	42
s5' 0-'c'/'b';'Templ''	43 Here if "a" zero.
1603'dprt'Templ''	44 Print one root and
cr'use'sl''	45 return to beginning.
	46
RU Block '	47 This block is called from
	48 line 31 and from line 38.
s7' daprt'cr4' 'x'2' 'uc2'='lcl''	49 Label second root.
s6' go to's0'''	50 Switch. End of Program.

D. Output

This a reprint of the results of the program at run time.

```
Roots of Quadratic Equation.

a = +1'+1'              1.000
b = +3'+1'              3.000
c = +2'+1'              2.000

Roots              Real          Imaginary
 x1 =             -1.000
 x2 =             -2.000


a = +1'+1'              1.000
b = +0'+0'               .000
c = +1'+1'              1.000

Roots              Real          Imaginary
 x1 =              .000                1.000
 x2 =              .000               -1.000


a = +0'+0'               .000
b = +1'+0'               .100
c = -2'+0'              -.200

Roots              Real          Imaginary
 x1 =             2.000


a =
```

CHAPTER III

ACT LANGUAGE

ACT Language consists of words of five characters or less, each word followed by a conditional stop code ('). These words may represent operators, C-words (constants), labels, or variables. Words are combined to form instructive statements, which in turn make up a program. Refer to the program in Chapter II for specific examples.

A. Characters and Controls

>character: A character is any numeric digit, letter of the alphabet, or any of the symbols - ; . , / or space. Since the space is a character, it must be used with care. Note the identity of the letter "L" and the digit "1". Note, also, that the computer cannot differentiate between upper and lower case symbols. The correspondence is obvious for the letters. For the digits and other characters, refer to the table in Appendix C, in which the pairs of characters in the columns "u.c." and "l.c." are indistinguishable .

>control: A control causes the flexowriter to perform one of the following: upper case, lower case, backspace, carriage return, tabulate, and color shift. It is not recommended that the tabulate be used in source programs except at the beginning of each statement, as shown in the program of Chapter II. Even this is not essential, but it is an aid to legibility.

B. Words

>word: A word consists of one to five characters plus controls (except tabulate) as desired, followed by a conditional stop code ('). e.g. a) The word Templ', line 26 consists of five characters (t, e, m, p, l) and two controls (upper case, lower case). b) The word a' line 9 is a one-character word.

>operator: An operator is any word taken from the list of operators, Appendix B. It can be defined loosely as a "command" to the computer. Individual operators are discussed separately in appropriate sections. In lines 35 and 36, for example, the operators are: sqrt', 0-', /', ;' and dprt'.

>C-word: A C-word is any word consisting of one to five integers, or consisting of "+" or "." followed by one to four integers. e.g. On line 25, .2' is a C-word.

>constant: A constant is a word or set of words representing a definite numerical value. The first (or only) word must belong to the class of C-words. A source-language constant is always positive, and the maximum number of such constants within a program is limited to 63.

The representation of a constant in the source program depends, as was mentioned in Chapter I, on whether it is a floating-point constant or an integer constant.

Integer Constants - Integer constants may consist of one word, or two words. Integers of five digits or less occupy one word of five digits (or less) followed by a conditional stop. Integers of six to nine digits occupy two words. The first consists of a plus sign (+) followed by one to four digits; the second word must be one to five numeric digits. Remember that the largest integer the system can handle is 536,870,911. e.g. On lines 10, 13, 16, etc. the word 1603' is an example of a one-word integer.

Floating-Point Constants - Floating-point constants always occupy four words. The first two words are the fraction. The first word consists of a decimal point followed by the first one to four digits of the fraction. The second

word contains the remaining digits of the fraction and is limited to five digits. If there are no remaining digits, this second word is a blank. The third word is e' if the exponent is positive and e-' if the exponent is negative. The fourth word is the absolute value of the exponent, one or two digits as though it were a one-word integer. Remember that although as many as nine digits may be given for the fraction, the internal form of floating-point numbers is such that they are rounded to roughly 3 in the 8th significant digit.

There are two floating-point constants in the program in Chapter II; .2''e'1' (=2.0) on line 25 and .4''e'1' (=4.0) on line 27. The constant 1.0 should be written as .9999'99999'e'0' (and similarly for other powers of 10) for most accurate representation. When written this way, it is represented as 0.99999994, whereas .1''e'1' is represented as 1.0000002.

Zero - The representation of zero is the one-word integer, 0', and is zero for both floating-point and integer arithmetic.

The table at the end of the chapter gives examples of source-language constants.

>labels: A label is word whose first character is "s", and whose remaining 1 to 4 characters are numeric digits, representing an integer from 0 to 190 inclusive. These are used to label statements. There is one exception to the form described above. This occurs when using procedures and is discussed in Chapter X, section E. On line 35, s2' is a label. Here the statement of line 35 is labeled "s2". On line 38, s6' and s7' are labels and refer to statements labeled "s6" and "s7" on lines 50 and 49 respectively. Note that s0001 is the same as s1.

>variable: A variable is any word of 5 characters or less which is not an operator, label, or C-word. Variables are discussed further in Chapter IX. There the definition is extended and the subscripting of a variable is discussed. The variables in lines 35 and 36 are: discr', 2a', Temp2', and Temp1'.

>blank: A blank is a word with no characters (conditional stop code only). Thus, the last two words of line 50 are blank words. A blank may include controls, as desired (except tabulate).

C. Statements

>statement: A statement consists of a sequence of words obeying the rules of syntax ending with a blank. The first statement of the program starts on line 4 and ends on line 5. The last statement of the program is on line 50.

>remark: A remark is any sequence of characters and controls (except tabulate) which is followed by a conditional stop and which obeys the following rule. It must contain at least six characters and the sixth character preceding the conditional stop must be one of the command letters, which are "tidybrazenchumps". A remark may be inserted in the program at any point; it will be ignored by the translator. If in such a sequence the sixth character preceding the conditional stop is any character other than one of the above, the sequence is treated as a blank. The words of lines 1 and 2 make up a remark.

D. Program

>program: A program consists of a sequence of statements. The last is followed by a (second) blank. Thus, the extra blank word after the statement on line 50 signifies that the preceding group of statements make up a complete program.

E. Other Definitions

Other terms which are conveniently defined at this time are:

>operation: This is computation which occurs at run-time (third phase) as a result of the inclusion of the corresponding operator in the source program (first phase). Notice the difference between ">operation" and ">operator".

>execution: This is the performance (in phase three) of the operation (s) associated with an operator, expression, statement, or program.

ACT-LANGUAGE CONSTANTS

SOURCE LANGUAGE	EQUIVALENT	COMMENTS
1'	1	integer
54321'	54321	integer
+12'34567'	1234567	integer
0'	0	integer or fl. pt.
.2''e'1'	2.0	floating-point
.5678'9'e'3'	567.89	floating-point
.9999'99999'e'0'	.99999994	The recommended form for fl. pt. "1.0" .

CHAPTER IV

OPERATORS - General Discussion

A. Introduction

Consider the statement on line 27 of the sample program in Chapter II, which is

```
b'x'b'-'.4''e'1'x'a'x'c';'discr''
```

This, of course, is the ACT III translation of the equation,

$$d = b^2 - 4ac \quad ,$$

where "d" is called "discr" because it is more mnemonic. The above statement is also an example of the arithmetic statement which is the basic working unit of any ACT program. Since the form of an arithmetic statement is quite straightforward, we will use it to illustrate many rules of syntax. In the first place ACT has no use for the "equal" sign, which expresses a passive statement of fact. It is replaced by the substitution operator ";" which has the meaning: store the expression on the left in the (symbolic) location specified on the right. The Operators "+", and "-", and "x" are also illustrated in this statement. The order of execution of these operations is the same as that understood in algebra. Thus "x" and "/" are executed before "-" and "+". Also note that the multiplication sign "x" must never be omitted in ACT language, as it may be in algebra. Consider, further, the equation,

$$y = (a + bc)/(d - e).$$

This equation when transcribed into ACT III language becomes the following statement:

```
['a'+'b'x'c'] '/' ['d'-'e'];'y''
```

Note the use of brackets in the same manner in which they are used in the original equation. They must, however, be square brackets. The operators used in the examples above all imply floating-point arithmetic. Therefore, the symbols a, b, c, d, e, y, and discr, are all floating-point variables.

In order to be more specific about the order of operations and about rules of syntax, we will need to make use of a few special concepts. They are the following.

>precedence: This is a number assigned to each operator to be used in determining the unambiguous meaning of a statement. In general, except for cases altered by brackets, operators of higher precedence are executed first.

>operand: This is a quantity upon which an operator acts; thus, in the combination b'x'c' the variables "b" and "c" are left and right operand, respectively, of the operator "x".

>expression: The simplest expression is a single variable. An expression is also any sequence of operators, each of which has all the operands which it requires. Thus, in the above example, b'x'c' is an expression. However, a'+'b' is not, since b'x'c' rather than b' is the right operand of +'.

>type: The type of an operator or expression indicates the way in which it may be combined with other operators. Certain capital letters are used to indicate "type". They are listed here, together with their explanation. In tables they are often followed by another capital letter; either "V" for variable or "E" for expression. Succeeding chapters discuss each type further.

F - Floating Point

I - Integer

L - Logic

X - No Result (This type may not appear in expressions used as operands).

The result, then, of the execution of a type F operator is a floating-point value, and, therefore, may be operated upon by another operation requiring a floating-point operand. The type of an expression is the type of its lowest ranking operator. (The discussion of "rank" follows).

B. Determination of Rank

The basic concept required to determine without ambiguity the exact meaning to the compiler of any string of words is the "rank". This concept extends the "precedence" idea to include the effect of brackets on the order of execution.

>rank: This is a number value which may be determined in the following way.

1) Assign the value zero to the reference quantity, "bracket level".
2) Starting from the left end, scan the statement, adding 4 to the bracket level for each left bracket "[", and subtracting 4 for each right bracket "]".
3) Then as the statement is scanned the rank of any operator is equal to its precedence plus the bracket level.

General Rule - With this definition of rank, we may state generally that, of two neighboring operators in a statement, the one of higher rank is executed first. If they have the same rank, the one on the left is executed first.

Rank may also be assigned to expressions and variables. The use of this concept can be seen in rules 2) and 3) of the next section.

4) The rank of any expression is equal to the smallest of the ranks of all the operators in the expression.

5) The rank of a variable is four plus the bracket level.

C. Basic Rules of Syntax

Although the General Rule, above, is adequate for many cases, it does not completely cover all situations. This section, therefore, gives a complete set of rules of syntax by which the unambiguous meaning of any statement can be determined.

1) The bracket level may never be greater than 28 nor less than zero. In other words, brackets may be nested to a maximum depth of seven. Redundant brackets are, as a rule, ignored. In any case, they can do no more harm than add a few milliseconds to the running time of the program. Thus, when the programmer is doubtful about precedence rules, it is suggested he play safe and insert brackets to remove possible ambiguity.

2) The left operand of any operator is determined by scanning left from the operator to, but not including, the first variable or operator of smaller rank than the operator in question, or the first expression of type X or L.

3) The right operand of any operator is determined by scanning right to, but not including, the first variable or operator of the same or smaller rank than the operator in question, or the first expression of type X or L.

4) Every operand must be an expression.

5) The type (F or I) of an expression used as an operand must agree with the type required by the definition of the operator. An expression of type X or L may not be used as an operand.

6) In general, two variables, or a variable and a constant, or two constants may not appear in the same statement without an intervening operator. The exception to this is in a statement beginning with one of the words - dim, index, dbind, enter, local.

Examples: Consider the sample statement used above,

['a'+'b'x'c'] '/'['d'-'e'];'y''

Here we observe the following relations:

operator	precedence	rank	left operand	right operand
;	0	0	All to its left	y
+	1	5	a	b'x'c'
x	2	6	b	c
/	2	2	['a'+'b'x'c']	['d'-'e']
-	1	5	d	e

Notice that the substitution operator has precedence zero, and therefore, its rank in the example is zero. Its left operand is the entire expression from the beginning of the statement. Note that, of course, it is meaningless for the right operand of ";" to be a complex expression; it must be the variable whose value is to be assigned.

CHAPTER V

ARITHMETIC STATEMENTS

This chapter discusses the arithmetic operators in detail (both floating-point and integer), as well as the more common transcendental functions which are included in ACT III language. Operators used to convert a numerical variable from floating-point representation to integer representation and vice versa are also explained here. At the end of the chapter, by way of summary, the operators discussed are listed in tabular form.

A. Floating-Point Arithmetic Operators

1. Basic Arithmetic Operators

As illustrated in the previous chapter the symbols representing the basic floating-point arithmetic operations are the same as that for algebra.

';' e.g. a';'b' The value of "a" is substituted into "b". ("a" is unchanged). The substitution operator has precedence "0". Its left operand is the entire expression from the beginning of the statement. The right operand must be a variable whose value is to be assigned. It is meaningless, of course, for the right operand to be a complex expression.

'+' e.g. a'+'b' Addition is precedence "1" and both right and left operands may be either an expression or a variable of type F.

'-' e.g. a'-'b' Subtraction is precedence "1" and both right and left operands may be either an expression or a variable of type F.

x' e.g. a'x'b' Multiplication is precedence "2" and both right and left operands may be either an expression or a variable of type F.

/' e.g. a'/'b' Division is precedence "2" and both right and left operands may be either an expression or a variable of type F.

0-' e.g. 0-'a' This is a unary operator which has only a right operand, and which has the effect of changing the sign of its operand (type F). It has precedence "3" which implies that any complex expression used as an operand for "0-" must be bracketed. The phrase a'x'['-'b']' is illegal because "-" has no left operand. It may, however, be corrected by writing it in the form a'x'0-'b'. No brackets are necessary because 0-'b' is unambiguously the right operand of "x".

abs' e.g. abs'a' This is also a unary operator, the operation of which produces the absolute value of its (right) operand (type F). It also has precedence "3" and therefore, any expression used as an operand must be bracketed.

2. Exponentiation

pwr' e.g. a'pwr'b' The operation "pwr" has the effect of raising the left operand (type F and positive) to the power of the right operand (type F). Thus a'pwr'b' gives a^b. The left operand (a) must be positive because the subroutine for "pwr" uses the relationship

$$a^b = \exp(b \ln a)$$

and, of course, the logarithm of a negative number is undefined. The operator "pwr" has precedence "3" which causes it to be placed in the order usually understood in algebraic expressions,

before multiplication and division. Consider as an illustration the coding of the equation,

$$w = a/(b-c^{-y^2})$$

The first attempt might produce,

a'/'['b'-'c'pwr'0-'['y'x'y']']';'w'' .

This is incorrect, as is seen by seeking the right operand of "pwr". Since "0-" has the same rank as "pwr", it is observed that "pwr" has no right operand. Thus, we must write

a'/'['b'-'c'pwr'['0-'y'x'y']']';'w'' .

Notice that small integral powers of a floating-point value are most efficiently obtained by direct multiplication and division. Thus, a'x'a' is better than a'pwr'.2''e'1'.

x10p' e.g. a'x10p'b' Change the exponent of a Floating-Point Number. This is the first operator introduced with one operand of type F and the other of type I. The operation of x10p takes the left operand (type F) "a" times 10^b where "b" is type I.

3. Common Functions

A number of single-valued floating-point functions are available in ACT III language. They are:

sqrt' e.g. sqrt'a' Square root of "a".

ln' e.g. ln'a' Natural logarithm of "a".

log' e.g. log'a' Common logarithm of "a".

exp' e.g. exp'a' Exponential of "a" (e^a).

sin' e.g. sin'a' Sine of "a (radians)".

cos' e.g. cos'a' Cosine of "a (radians)".

artan' e.g. artan'a' Arctangent of "a". (Radians, $-\pi/2$ to $+\pi/2$.)

These are all unary operators (right operand only) of precedence 3. This gives them the rank normally understood in algebraic expressions. Again any complex expression used as the right operand must be bracketed. Syntactically then, the above operators plus "0-" and "abs" are in the same category.

4. Random Number

randm' The operator "randm" has no operands, but produces at each use a new member of a set of pseudo-random floating point numbers, uniformly distributed over the interval from 0 to 1. A few examples of some uses are: randm'/'randm';'y'' (The quotient of two random numbers stored in "y"); sin' ['randm']'' (The sine of a random number between 0 and 1.). Note: "randm" has precedence 3.

B. Integer Arithmetic Operators

As was stated earlier, the largest integer which can be handled in ACT III is 536, 870, 911. The most common use of integers is for counting and subscripts (see Chapter VII). Under these conditions they are not expected to become larger than a few thousand (positive or negative). As a general rule, integer operators begin with the letter "i".

1. Basic Arithmetic Operators

;' e.g. a';'b' Substitution. This is the same operator as was described under arithmetic floating point operators. (See A.1.) Since it works equally well with integer and floating point values the programmer must keep track of the type of expression or variable involved.

i+ ' e.g. a'i + 'b' Machine Addition

i-' e.g. a'i-'b' Machine Subtraction

Addition and subtraction are machine operations in which overflow will occur if the result exceeds (±) 536, 870, 911. On most machines this will cause a stop. On machines equipped with overflow logic modification it will result in an incorrect answer and set a special indicator which may be tested later (see Chapter XI). These are precedence "1"; both left and right operands of each may be either an expression or a variable of type I.

ix' e.g. a'ix'b' Integer Multiplication, with Error Stop

This operation will yield correct answers if the answer is not greater (in magnitude) than 536, 870, 911 and will cause an error stop if this limit is exceeded. It has precedence "2", and both right and left operands may be either an expression or a variable of type I.

nx' e.g. a'nx'b' Integer Multiplication

If small numbers are being handled, and there is no danger of overflow, integer multiplication is performed much faster by the use of this operator than by the use of the above operator "ix". However, if the answer would exceed 134, 217, 727 in magnitude, "nx" will yield an incorrect answer without any warning. This operator is also precedence "2" and both right and left operands may be either an expression or a variable of type I.

i/' e.g. a'i/'b' Integer Division

Some special consideration is necessary in integer division, since the exact quotient of two integers may not be an integer. The operation of "i/" is best described with the help of the formula

$$n = qd + r$$

where n, d, q and r (all integers) are the numerator, denominator, quotient and remainder respectively. For a given n and d, the choice of q and r is not unique. We use the convention that r must have the same sign as d and be less than d in absolute value. This is equivalent to the statement that q is the (algebraically) largest integer which is (algebraically) less than or equal to the exact value of n/d. The answer returned by the operation of "i/" is q. The value of r is stored in the symbolic location "remdr", where it is available for use immediately or later in the program. Consider the following examples.

n =	10	11	-10	-11	-11
d =	5	5	5	5	-5
q =	2	2	-2	-3	-3
r =	0	1	0	4	-4

iabs' e.g. iabs'a' Integer Absolute

The operation of "iabs" gives the absolute value of the right operand (type I). Its syntax is similar to its floating point counterpart "abs".

ipwr' e.g. a'ipwr'b' Integer Power

The operation of "ipwr" gives the value of a^b as an integer. If b = 0 the answer is 1. If a = 0 and b is negative an error indication is returned. If a = 1 and b is negative the answer is zero. For all other cases the answer is exact. An error indication is returned if $a^b \geq 2^{29}$. The operator "ipwr" has precedence "3" and both of its operands must be expressions of type I.

C. Mixed Arithmetic Operators

These operators are used to change a variable from its integer form to its floating-point form and vice versa. These operators are all of precedence "3".

flo' e.g. a'flo'b' Change"b" (type I) to a Floating-Point Number. Both left and right operands are type I. The operation of "flo" inserts a decimal point in the right operand to space off the number of fractional digits specified by the left operand, and expresses the resulting number ($b \times 10^{-a}$) in floating-point form.

unflo' e.g. a'unflo'b' Change "b" (type F) to an Integer With Rounding. This operator is essentially the reverse of "flo" . The right operand (type F) is multiplied by 10^a, where "a" is of type I and the resulting number is rounded to the nearest integer.

fix' e.g. a'fix'b' Change "b" (type F) to an Integer Without Rounding. The syntax of the operator "fix" is the same as that for "unflo". In the operation, however, the result is not rounded, fractional digits being dropped without rounding.

The following examples illustrate the operation of "flo", "unflo", and "fix".

Left Operand (Value)	Operator	Right Operand (Value)	Result
0	flo	50	$.5 \times 10^2$
3	flo	-1234	$-.1234 \times 10^1$
-1	flo	15	$.15 \times 10^3$
0	unflo	$.1357 \times 10^1$	1
1	unflo	$\pm.1357 \times 10^1$	14
1	fix	$.1357 \times 10^1$	13

D. Tables

Floating-Point Operators

Name	Type	Left Operand	Right Operand	Precedence	Restrictions
;	F	FE	FV	0	
+	F	FE	FE	1	
-	F	FE	FE	1	
x	F	FE	FE	2	
/	F	FE	FE	2	Rt. Op. $\neq 0$.
pwr	F	FE	FE	3	L. Op. >0.
x10p	F	IE	FE	3	
0-	F	None	FE	3	
abs	F	None	FE	3	
sqrt	F	None	FE	3	Rt. Op. ≥ 0.
ln	F	None	FE	3	Rt. Op. > 0.
log	F	None	FE	3	Rt. Op. > 0.
exp	F	None	FE	3	Rt. Op. $< 13 \cdot 897423$
sin	F	None	FE	3	$\lvert$Rt. Op. $\rvert < 10^8$
cos	F	None	FE	3	$\lvert$Rt. Op. $\rvert < 10^8$

Integer and Mixed Operators

Operator	Type	Precedence	Left Operand	Right Operand	Comments
;	I	0	IE	IV	(also F)
i+	I	1	IE	IE	
i-	I	1	IE	IE	
ix	I	2	IE	IE	
i/	I	2	IE	IE	
nx	I	2	IE	IE	
flo	F	3	IE	IE	
unflo	I	3	IE	FE	
fix	I	3	IE	FE	
iabs	I	3	None	IE	
ipwr	I	3	IE	IE	

D. Tables

Floating-Point Operators

Name	Type	Left Operand	Right Operand	Precedence	Restrictions
;	F	FE	FV	0	
+	F	FE	FE	1	
-	F	FE	FE	1	
x	F	FE	FE	2	
/	F	FE	FE	2	Rt. Op. $\neq 0$.
pwr	F	FE	FE	3	L. Op. >0.
x10p	F	IE	FE	3	
0-	F	None	FE	3	
abs	F	None	FE	3	
sqrt	F	None	FE	3	Rt. Op. ≥ 0.
ln	F	None	FE	3	Rt. Op. > 0.
log	F	None	FE	3	Rt. Op. > 0.
exp	F	None	FE	3	Rt. Op. $< 13 \cdot 897423$
sin	F	None	FE	3	$\lvert$Rt. Op.$\rvert < 10^8$
cos	F	None	FE	3	$\lvert$Rt. Op.$\rvert < 10^8$

Integer and Mixed Operators

Operator	Type	Precedence	Left Operand	Right Operand	Comments
;	I	0	IE	IV	(also F)
i+	I	1	IE	IE	
i-	I	1	IE	IE	
ix	I	2	IE	IE	
i/	I	2	IE	IE	
nx	I	2	IE	IE	
flo	F	3	IE	IE	
unflo	I	3	IE	FE	
fix	I	3	IE	FE	
iabs	I	3	None	IE	
ipwr	I	3	IE	IE	

CHAPTER VI

FLOW CONTROL

A. Normal Flow

By the term "flow" we understand the order of execution (in phase 3) of the statements in a program. The normal flow is the order in which the statements appear in the source program, beginning with the first, and stopping after execution of the last statement.

B. Labels and Labeled Statements

In order to make it possible to modify the flow, it is necessary to be able to label a statement. Such labels must consist of the letter "s" followed by one to four numeric digits representing an integer less than 191.

If the first word of a statement (exclusive of any remarks) is a label, the statement is assigned this label. The labeling word is not included in the syntax of the statement. It is customary to refer to the statement labeled (for example) s53 as statement number 53.

No two statements may bear the same label in any program.

With this introduction of labels into the normal flow of the program, we recognize labels as allowable operands of certain operators. This will be indicated in the operator tables by the symbol L in the appropriate column. All labels appearing as operands must be assigned to statements in the program.

C. Logic Operators

We now introduce a class of operators which we call logic operators whose function is, in general, to control the flow of a program. Logic operators, like type X operators, may not appear in operands. They usually require labels

as operands. They are denoted as type L in the tables. Operators of this class have precedence zero.

D. Unconditional Transfer

The operator "use" directs the flow to the statement whose label is the right operand.

E. Conditional Transfer -- "if" Statements

The statement

```
if 'a'neg's10'zero's11'pos's12'
```

directs the flow to statement 10, 11 or 12 (these labels are used only as samples) according to the value of "a" (negative, zero or positive respectively). The variable "a" (used here only as a sample) may be replaced by any expression, either integer or floating-point.

Syntactically the operator "if" has precedence zero and has the effect of evaluating its right operand and leaving the value for use as the "previous result". Each of the test operators neg, zero and pos causes a test of this "previous result" and transfers to the statement whose label is its right operand if the condition is satisfied. It is not necessary to include all the tests, as long as those included are in the same order as indicated above. If none of the tests is satisfied, the normal flow (to the next statement) occurs.

Because of the problems of round-off and conversion errors, the "zero" test is not recommended for floating-point numbers, except to prevent a division by zero or related computational error.

F. Switches (Variable Transfers)

The statement

```
go to's0''
```

(which must be labeled if it is to be used) is a switch or variable transfer which may be changed in several ways. The symbol "s0" is not to be interpreted as a label, but rather as a part of the composite operator go to's0'. Note the space in "go to". Operators which require a switch as operand are designated in the tables by the symbol S in the operand column. The label of a switch may also be used in any way allowed for ordinary labels.

The statement

set's20'to's15''

when executed during the flow of the computation, sets the switch labeled s20 to the statement labeled s15. When the switch s20 is reached during the subsequent flow, it will transfer to statement 15 unless the setting has been changed in the meantime. If a switch is reached at a point in the flow before it has been set, it will transfer to itself, thus setting up a tight loop.

The statement

ret's20'use's15''

when executed during the flow of the computation, first sets the switch labeled s20 to the statement immediately following this one (which need not be labeled), then transfers to statement 15. This enables the effective insertion at different places in the computation of a block of statements beginning with statement 15 and ending with the switch.

G. Loops -- "for" Statements

The execution of the statement

for'i'step'j'until'n'rpeat's20'

has the following effect. First "i" (which represents any integer variable) is incremented by the amount "j". If this step carries the value of "i" past the value of "n", control flows to the next statement. Otherwise, control flows to statement 20. Note: The magnitude of j(n-i) must not exceed 134,217,727.

Here "j" represents any integer expression, which may be positive or negative. If the value of "j" is zero, control always flows to the next statement. Also "n" represents any integer expression, and "s20" represents any label.

The simplest application of this statement is for programming loops. Thus sample program No. 1 in Appendix C causes the reading of an integer "n", followed by the reading of "n" sets of values of two floating-point numbers "a" and "b", printing for each set the values of a and b and their sum, difference, product and quotient. The test for zero value of "b" prevents a division by zero.

We do not recommend the use of floating-point numbers for loop control; however, the equivalent to this statement in floating-point arithmetic may be written using the standard floating-point operators and conditional transfer tests.

LOGIC OPERATORS

Operator	Type	Precedence	Left Operand	Right Operand	Comments
use	L	0	None	L	
if	X	0	None	E	
neg	L	0	None	L	
zero	L	0	None	L	
pos	L	0	None	L	Must not occur before "neg" or "zero".

LOGIC OPERATORS CONT.

Operator	Type	Precedence	Left Operand	Right Operand	Comments
go to's0'	X		None	None	Must occupy whole statement
set	L	0	None	S	
to	L	0	None	L	
ret	L	0	None	S	
for	X	0	None	None	
step	I	0	IV	IE	
until	X	0	IE	IE	
rpeat	L	0	None	L	

H. Stopping the Computer

The operator "stop", when executed at run-time, causes the computer to stop. If it is restarted, the program will continue in sequence. The operator "wait", followed by one conditional stop (not two) is placed on a source program tape to suspend compilation. Pressing "start" continues compilation.

CHAPTER VII

INPUT AND OUTPUT

A. The Input-Output Problem

The information contained in the preceding chapters is sufficient for the writing of solutions to many problems; however, no means has been provided for reading data or printing answers. In order to make it possible to write working programs at the earliest possible moment, this chapter will discuss the most commonly used methods of input and output. At this point it is necessary to make an exception to our basic philosophy of ignoring the mechanics of the computer at the programming stage.

Numeric data may be read from punched paper tape, either through a Flexowriter (in which case the data are printed as they are read) or through a photoelectric reader (without simultaneous printing). Output may be taken on the same Flexowriter (with or without a punched tape copy) or on the high-speed punch (which produces a tape which must subsequently be listed on a Flexowriter). In any case, data tapes must be initially prepared, and output must be eventually printed, on a Flexowriter. If the Flexowriter is to be used directly for both input and output, both will appear on the same "hard copy". If any other option is used, the input data will not appear in the output unless the program specifically provides for it to be printed out. Since the output format is, because of hardware limitations, much more flexible than the input, the latter method is usually to be recommended. However, the choice may be limited by facilities or local operating procedures at each installation. We will, therefore, not discuss this matter further.

B. Numeric Input

B. 1. Floating Point Data Input

The execution of the operator "read" effects the reading of one floating point number and its storage in the location of the right operand, which must be a

variable.

The number to be read by "read" must be punched on the data tape in the following format. First the fraction is punched as a sign (+or-) followed by one to seven digits of the fraction. The decimal point must not be punched, but it is understood to be immediately after the sign. The sign must be the first character read; however, controls (except tab) may be inserted to improve legibility. If the full seven numeric digits are punched, any desired descriptive material (not containing an apostrophe or stop code) may precede the sign. The fraction is followed by a stop code ('), and then the exponent is punched as a sign (+ or -) and one or two digits followed by a stop code.

The following examples will illustrate the format for floating point data.

Data Tape Contents	Value
+1'+1'	0.1×10^{1}
+1000000'+1'	0.1×10^{1}
-53'+0'	-0.53×10^{0}
+123'-15'	0.123×10^{-15}
+1'-35'	0 (too small)
+1'+35'	error (too large)

B. 2. Integer Data Input

The execution of the operator "iread" effects the reading of one integer and its storage in the location of the right operand, which must be a variable.

The number to be read by "iread" must be punched on the data tape in the following format. The first character must be the sign (+ or -), followed

by one to seven digits, as required, and followed by a stop code. Zeros or spaces may be inserted immediately following the sign if desired, provided the total number of spaces and digits does not exceed seven. A positive integer may be preceded by any textual material ending with seven spaces. In this case, do not punch the + sign. Controls (except tab) may be inserted as desired.

The following examples illustrate the format for integer data.

Data Tape Contents	Value
+1'	1
-25'	-25
+003'	3

B. 3. Read and Float

Numbers may be read in integer form and converted to floating point by the use of the operator "rdflo". To give a specific illustration, the statement

```
n'rdflo'a''
```

has the same effect as the two statements

```
iread'a''
n'flo'a';'a''
```

Here the left operand ("n") must be an integer expression, and the right operand ("a") must be a variable. The value stored in "a" is floating point. A convenient way of thinking of it is that a decimal point is understood "n" digits to the left of the stop code (to the right if "n" is negative). Thus, the following examples are formulated.

"n"	Tape Contents	Value
0	+12'	0.12×10^2
1	-12'	-0.12×10^1
-1	+12'	0.12×10^3

B. 4. Syntax of Input Operators

The three input operators, "read", "iread", and "rdflo" are all precedence zero, and may be used in operands. Thus, the statement

```
read'a';'b';'c''
```

will produce the operation of reading a floating point datum and storing it in a, b, and c. Also the statement

```
['read'a']'x'b';'c''
```

effects the storing of the number read into "a", and the product a x b into c. The brackets are necessary for the proper rank relationships.

B. 5. The Input Switch

Normal flow of control ordinarily occurs after execution of an input statement. If,however, a blank (stop code only) is read on the data tape, control is transferred to a switch internal to the package. This switch is set by execution of the expression

```
rdxit's10'
```

to the statement labeled (for example) s10.

This may be used, for example, to signal the end of a data block, or (on unattended runs with many sets of data) to suspend computation. The input switch, like program switches, retains a setting until it is changed in the flow of the computation.

C. Numeric Output

C. 1. Format Control for Numeric Output

The output operators to be described next all require as left operand an integer (expression) which will be referred to as the format control. The value "n" of this operand is considered to have the form

$$n = 100c + f$$

where c and f are positive integers and f is less than 100. The general effect of this quantity may be described in the following way. The output number is printed in a field "c" characters wide, with "f" fractional digits, and leading spaces as needed to move the output to the extreme right within the specified field. This gives complete flexibility in the arrangement of output numbers on a single line. The "sign" referred to in each discussion is a space for positive numbers, "-" for negative numbers.

Line spacing is controlled by the operator "cr", which controls the typewriter operation "carriage return". The operator "tab" is available to control the typewriter operation "tabulate".

C. 2. Integer Output

The execution of the operator "iprt" causes the value of the (integer) right operand to be printed under format control of the left operand as follows, where the integer to be printed has "d" digits.

(a) If f is zero, the operation involves the printing of c-d-1 spaces (none if this is negative), then the sign (space or "-"), then the "d" digits.

(b) If $0 < f < d$, the output will consist of c-d-2 spaces, the sign, d-f digits, a decimal point, and "f" digits.

(c) If $d \leq f < 9$, the output will consist of c-f-3 spaces, the sign, one zero, a decimal point, f-d zeros, and d digits. If f = 8, the last digit may be in error by one unit. A value of f greater than 8 should not be used.

C. 3. Floating Point Output

The execution of the operator "print" causes the value of the (floating point) right operand to be printed under format control of the left operand as follows.

(a) If $c > f + 7$, the output consists of c-f-7 spaces, sign of number , decimal point, f significant digits of fraction (rounded to last printed digit), space, the letter "e", sign and two digits of exponent.

(b) If c is less than f + 8 but greater than 7, the output is as for (a) but with f replaced by c-7.

(c) If c is less than 8, the output is as for (b) but with c replaced by 7.

C. 4. Unfloated Output

The execution of the operator "dprt" (short for decimal print) causes the value of the (floating point) right operand to be printed under format control of the left operand as a decimal number in common decimal form. In the specific description given below, the exponent of the number to be printed is denoted by "e".

(a) If e is greater than zero and c is greater than e + f + 1, the output consists of c-e-f-2 spaces, sign, "e" integral digits, decimal point, f fractional digits (rounded to the last digit printed).

(b) If e is greater than zero and c is less than e + f + 2, the output is as for (a) but with f replaced by c-e-2, or, if this is negative, by zero.

(c) If e is not greater than zero, the output is as for (a) or (b) above, but with e replaced by zero.

Some illustrative examples will be found in Appendix B.

C. 5. Syntax of Output Operators

The operators "iprt", "print", and "dprt" are all precedence zero and type X. It may be deduced from the rules of syntax that a single statement may not include more than one of these operations unless they are separated by an expression of the same rank. Thus, the statement

1608'print'a'cr'1608'print'b''

is satisfactory, but if the "cr" were omitted it would be incorrect.

We note again here that the limit of internal accuracy for floating point numbers is ±3 in the eighth significant figure, and, therefore, printed results cannot be relied on beyond this accuracy.

D. Alphabetic Input and Output

D. 1. Direct Alphabetic Print "daprt".

The operator "daprt" effects the printing of any alphabetic information, giving the programmer direct control over all operations of the output Flexowriter. Thus, for example, the statement

```
daprt's't'o'p''
```

will cause the compilation of instructions to print the word "stop" (at run-time). Up to 63 characters may be so indicated in a single statement; if more are required, additional "daprt" statements may be included. Each character to be printed (including space) must occupy a separate word. The controls are represented by a set of mnemonic codes, each containing at least two characters, as follows.

lower case	lc1'
upper case	uc2'
color shift	color'
carriage return	cr4'
backspace	bs5'
conditional stop	stop'
(or apostrophe	ap')
tab	tab6'
nonprinting stop codes:	
conditional	stop9'
unconditional	stopu'

(The last two codes above are for use only when output is on the high speed punch. When the tape so punched is listed, these codes stop the Flexowriter without printing; the first of the two is dependent on the conditional stop

button, whereas the second is not. If output is on the Flexowriter these instructions are ignored.)

D. 2. Repeated Print "reprt".

The execution of the statement

IE'reprt'C''

(where IE means any integer expression, and C stands for any character or control, as discussed in section C. 1. above) causes the high-speed printing of the indicated character or control "IE" times. If "IE" is negative or zero, nothing is printed. If "IE" is less than 6, it is more efficient to use "daprt".

D. 3. Alphabetic Read "aread".

The execution of the statement

aread'V''

causes a single word to be read from tape, interpreted as one to four alphameric characters in a special two-digit code, and stored in location "V". The two-digit code is given in Appendix A.

A blank word causes a transfer to the input switch.

D. 4. Alphabetic Print "aprt".

The execution of the statement

aprt'V''

causes the printing of five characters (and/or controls) or less. The contents of "V" may have been fed in as data to "aread", in which case a maximum of four characters may be printed. If "V" is negative, nothing is printed.

Note: Users wishing to do so may code alphabetic data in hex, five 6-bit characters in bits 1-30. If less than five characters are used, the remaining 6-bit fields

(at the right) are left blank. Such data may be read in using "rdhex" (see below).

E. Compatible Input-Output.

It is frequently desirable to use the output of one program as input to another. Several forms are available.

The output of "punch" and "ipch" is easily read and may be interspersed with other forms of output (see preceding sections). The other output, unless it contains stop codes from "daprt", is ignored on read-in.

E. 1. Floating Point "punch".

The execution of the expression

punch'FE'

causes the value of "FE" to be punched (printed) in the following form. The fraction is rounded to seven digits and punched as sign, seven digits and stop code. The exponent is punched as sign, two digits and stop code. This may be used as input to "read".

E. 2. Integer "ipch".

The execution of the phrase

ipch'IE'

causes the value of "IE" to be punched as a sign, seven digits (with leading zeros as needed) and a stop code. This may be used as input to "iread".

E. 3. Use of "iprt".

The execution of the statement

0'iprt'IE'daprt'stop''

causes the value of "IE" to be punched as a sign, significant digits (no leading zeros) and a stop code. This may also be used as input to "iread". However,

it must not follow noncompatible output forms directly.

E. 4. Hexadecimal Output "hxpch".

The execution of the phrase

hxpch'E'

causes the value of "E" to be punched as an eight-character hexadecimal word with stop code. This is not easily interpreted as are the forms previously discussed; however, it has certain advantages. In the case of floating point values, it is not necessary to round to seven digits, and thus greater accuracy is obtained. In the case of integer values, there is no restriction to seven-digit integers as in "ipch"; the full available range may be used.

E. 5. Hexadecimal Input "rdhex".

The execution of the phrase

rdhex'V'

causes a single hexadecimal word to be read and stored in "V". This may be used to read output from "hxpch", or numeric or alphabetic data handcoded in hex. A blank word is stored as blank; the input switch is not associated with "rdhex".

E. 6. Syntax of Compatible Input - Output.

The operators "punch", "ipch', "hxpch" are all of precedence zero, and require a right operand only. The operator "rdhex" is similar to "read" and "iread" in its syntax.

Table of Input-Output Operators

Name	Type	Left Operand	Right Operand	Precedence	Restrictions
read	F	None	FV	0	
iread	I	None	IV	0	
rdflo	F	IE	FV	0	
aread	A	None	AV	0	"A" = alphameric
rdhex	F, I, A	None	V	0	
print	X	IE	FE	0	
iprt	X	IE	IE	0	
dprt	X	IE	FE	0	
punch	X	None	FE	0	
ipch	X	None	IE	0	
aprt	X	None	AE	0	
hxpch	X	None	E	0	
daprt	X	None	*	0	*string of characters
reprt	X	IE	C	0	C = character.
RDXIT	L	None	L	0	
cr	X	None	None	0	
tab	X	None	None	0	

CHAPTER VIII

INTERMISSION

At this point enough of the vocabulary and syntax of ACT language has been presented to make possible the writing of meaningful programs. A few simple examples have been discussed briefly in previous chapters. We now consider some sample programs in greater detail.

A. Sample Program No. 2--Mean and Standard Deviation

For a set of numbers $y_1, y_2, \ldots, y_n$, the mean $\bar{y}$ and standard deviation σ are given by the formulas

$$\bar{y} = \frac{\sum y}{n}$$

and

$$\sigma = \sqrt{\frac{\sum y^2}{n} - \left(\frac{\sum y}{n}\right)^2}$$

where $\sum y$ and $\sum y^2$ stand for the sums $y_1+\ldots+y_n$ and $y_1^2+\ldots+y_n^2$ respectively.

Sample program "2" reads a set of numbers y, expressed as integers, floats them and accumulates the two sums as the values are read. The input switch, called by a blank on the data tape, causes computation and printing of $\bar{y}$ and σ, unfloated to two decimal places, starting at statement 5. If the value of n is zero (an empty set), the program stops at statement 10; otherwise, new sets of data are read and processed. Note the use of "n" as a counter. When the blank is read, "n" gives the number of data words which have been read in the current set. Before this can be used in the floating point formulas, it must be converted to a floating point number.

There is the possibility, when this formula is used, that accumulated roundoff errors may cause the quantity under the root sign to be negative. This implies a small value of σ, and the program replaces it by zero in s6. A more accurate method in this case is contained in sample program no. 4, which is discussed after the introduction of subscripts.

B. Sample Program No. 3

This sample program is designed to illustrate the use of "ret--use" with a switch. The block of coding so called accomplishes the printing of the value of y, together with its sine and cosine. The main program reads two numbers a and b, then uses the block to treat first a, then b, then a + b, then a - b as y.

CHAPTER IX

REGIONS AND SUBSCRIPTED VARIABLES

A. Blocks of Data

There are many situations in which we desire to treat data in blocks, using a single name for the entire block and calling individual positions in the block by a serial number. Such blocks are reserved by the special operator "dim" (for "dimension"). The statement

dim'a'10'b'55''

as an example, reserves a block of 10 words under the name "a" and a block of 55 words under the name "b". Any number of blocks may be reserved by a single "dim" statement, and any number of "dim" statements may be used in a program, the only restriction being that the variables must appear in the statement before they are used in any other way. (Here and in similar r appearing in later sections, the relation "before" is to be interpreted as physical order of statements in the source program).

B. Integer Subscripts

When a variable name is followed immediately by a one-word inte combination is treated as a variable. The ten words reserved, in the abc example, for block "a" may be referred to as a'0', a'1', ...a'9'. It i assumed that "a" has previously been named in a "dim" statement before i so used. If "a" appears without a subscript, the subscript zero is underst Thus, the first word of block "a" is either a' or a'0'.. In the above exampl variable a'10' refers to the first word after a'9', which is b'0'. Blocks reserved

in a single dimension statement may thus be regarded as segments of a single larger block, if this is useful in a particular application.

C. Variable Subscripts

The statement (example)

```
index'i'j'k''
```

defines the variables i, j, k as subscripting variables. As many as 31 variables may be so defined in a single "index" statement, and any number of "index" statements may appear in a program. An "index" statement may appear at any point in the program, as long as the variables it contains have not appeared in the portion previously compiled.

The combination a'i' appearing in a program, where "i" has been named in an "index" statement, then refers to the (i+1)'th word of block "a". The value given to i is that which has most recently been assigned in the flow. Thus if i = 2, a'i' is equivalent to a'2'. Subscripts must have integer values. If a subscript has not been properly set before it is used, the execution of the program will go astray.

Thus, to read 20 words into the locations beginning with a'1', we may write

```
        dim'a'21''
        index'i''
        1';'i''
s1'     read'a'i''
        for'i'step'1'until'20'rpeat's1''   .
```

D. Arithmetic in Subscripts

The combinations a'i'2' and a'2'i' (for example), where "i" has appeared in an "index" statement and "2" is a sample of any one-word integer, represents a variable, the effect being to add 2 to the value of "i" to obtain the value of the subscript.

E. Double Subscripts (twoscripts)

The statement (example)

dbind'm"

reserves a block of two locations (m'0' and m'1') and defines the variable "m" as a twoscript. Restrictions on "dbind" statements are the same as those on "index" statements. (The code "dbind" stands for "double index"). The use of double subscripts is best explained with the help of an example. Suppose the value of m' (i.e. m'0') has been set to "i" (an integer), and m'1' has been set to "j". Suppose, further, that the value of a'0' has been set to "n".

The variable a'm' then refers to the word in the block "a" whose subscript has the value

$$(i-1)n+j.$$

Thus, if the words in block "a", starting with a'1', are thought of as being arranged in a rectangular array by rows, with "n" columns ("n" being the current value of a'0'), the variable a'm' refers to the word in the i'th row and the j'th column of this array.

The following coding may be used to cause the printing of the elements of an array of 3 rows and 4 columns in a block.

```
                dim'a'13''
                dbind'm''
                4';'a'0''
                1';'m'0''
s1'             cr''
                1';'m'1''
s2'             1608'print'a'm''
                for'm'1'step'1'until'4'rpeat's2''
                for'm'0'step'1'until'3'rpeat's1''
```

Combinations such as a'm'2' (where "m" is a twoscript) should be avoided. They may be used, if the user is careful to observe that the number of columns must now be in a'2' (in this example), and the array starts with a'3'.

F. Some Definitions

>subscript: a variable which has been named in an "index" statement, and which has an integer value. These will be represented by the symbol S.

>twoscripts: a variable which has been named in a "dbind" statement. These will be represented by the symbol D.

>variable: (extended definition)

a) any word which is not an operator, label, or C-word.

b) two words, of which the first is not an operator, label, C-word or S, and the second is a one-word integer or S or D.

c) three or more words, of which the first is not an operator, label, C-word or S, one (or none) of the following words is S or D, and the remaining words are one-word integers.

(Note: the effect is to add all the integers if more than one is written. This is allowed, but not useful.)

The reader is advised to recall rule number 6 from Chapter II, Section 3, and reinterpret it with this new definition of "variable". For convenience in reference, we may use the term "simple variables" to denote those falling under class (a) above, and "subscripted variables" for those of class (b) or (c).

G. Subscripted Labels

The label concept may be extended also by the use of subscripts. Thus, the word-pair s1'i' (for example,) where "i" is a subscript, is treated as a label. The statement immediately before number 1 must consist of a series of "use'label" pairs. This is called a transfer vector .

Thus, for example, if the Coding is

use's5'use's20'use's3'use's75''s1' (etc.)

then the label s1'i', for the values 0, 1, 2, 3, 4 of "i", will be equivalent to s1, s75, s3, s20, s5 respectively.

Note the reverse order here.

>transfer vector: a single statement consisting of a series of "use'label" pairs. The labels in a transfer vector must not be subscripted. The "label" of a transfer vector is that of the following statement. The "length" of a transfer vector is the number of "use'label" pairs it contains.

The rules concerning subscripted labels may now be summarized as follows:

a) a subscripted label refers to an element of the transfer vector so labeled. The value of the subscript (which must not be greater than the length of the transfer vector--this condition must be insured by the

programmer) determines the element as counted backwards from the last element.

b) if the value of the subscript is zero, the reference is to the statement following the transfer vector.

c) a label with a variable subscript must not be used where a switch "S" is specified as an operand, or in a "ret-use" or "set-to" statement.

CHAPTER X

PROCEDURES

A. The Sub-Program Concept

The ability to write sub-programs which may be used at different points in a program, or in different programs, is a valuable one. A simple method of doing this has been described in Chapter IV, Section 6. This technique, which involves the "ret.....use" statement, has certain disadvantages. Thus, it is necessary for the user to be aware of a number of details of the sub-program, including statement numbers and variable names, in order that he can avoid duplication; also, this scheme is not very flexible when large blocks of data must be referenced. Thus, for example, a matrix inversion routine might be written which would invert the matrix "mat 1" and place the inverse in "mat 2", and this would be suitable as long as the user labels the two blocks with these names in his program; this, however, would be impossible if the routine were required to operate on several matrices in the same program, unless the user resorted to moving the data to and from the designated blocks. Thus, the "ret-use" sub-program is useful in the informal situation where a certain program segment can be called efficiently from two or more parts of a specific program.

B. The Procedure as a Sub-Program

The term "procedure" is used here to refer to a more formal type of sub-program which overcomes the drawbacks listed above. In particular, both statement numbers and variable names used within the body of a procedure are "local", in the sense that their definition is erased after compilation of the procedure body, thus removing all problems associated with duplication of

symbols. In addition, at each use of the procedure, the main program specifies the location of all data required and the location in which output information is to be stored. These locations are referred to within the body of the procedure by dummy symbols, thus making it possible to use the same procedure to operate on different blocks of data.

Procedures may be called on from within other procedures, and this process may be stacked to any depth. Thus this becomes one of the most powerful features of the ACT language.

As examples of the utility of the procedure concept, we refer to a number of procedures described and listed in Appendix. At this point, however, we will refer to one procedure, the details of which involve a wider understanding of LGP-30 coding than is given in this manual. This is the "MOVE" procedure. This procedure makes full use of the special assembly features of ACT III, together with a knowledge of the details of the object program, to accomplish the transfer of a block of data from one location to another as rapidly as the LGP-30 hardware allows. However, all that the user needs to be aware of is the calling sequence. Thus, the execution of the statement

```
if'n'call'move'arg'a'arg'b''
```

causes the rapid transfer of the "n" words starting at a'0' to the "n" locations starting at b'0', allowing for the possibility of overlap of the two blocks.

C. Procedure Operators

The statement

```
enter'name'a'b''
```

denotes the entry point of the procedure "NAME", and assigns the formal parameters

"a" and "b" to represent two main-program variables (or labels) named in the calling statement. As many as 31 formal parameters may be used in one "enter" statement. These constitute the formal parameter list of the procedure.

The execution of the statement

call'name'arg'ex'arg'y''

causes a transfer of flow to the procedure "NAME". The list of variables (and/or labels) punctuated by "arg" is used by the procedure to interpret the formal parameters appearing in the "enter" statement. This is the actual parameter list.

These two lists must agree in number of entries, and it is the responsibility of the programmer to see that they agree in order and type.

The formal parameters in the "enter" statement must be nonsubscripted variables. The elements of the "arg" list may be variables (simple or subscripted) or labels; they may not be switches or expressions containing operators.

The name of a procedure must not appear in a program before it is defined in an "enter" statement.

The execution of the statement

exit''

transfers the flow to the statement following the ''call'' statement from which the procedure was entered.

The statement

end''

denotes the physical end of the procedure body. When this statement is compiled, the statement dictionary is erased and all symbols whose first appearance was after the name of the procedure in the "enter" statement are erased from the symbol table.

D. The Procedure Body

A procedure body begins with an "enter" statement and is ended by the next "end" statement. No other "enter" statement may occur within a procedure body.

An "end" statement may not appear without a matching "enter" statement. An "exit" statement may not appear outside of a procedure body. Note that the "end" statement is merely an instruction to the compiler, and does not cause a return to the main program.

Computation in any program commences at the statement following the last "end" statement.

Since the "end" statement causes the statement dictionary to be erased, every label used before an "end" statement must be defined before that "end" statement.

E. The Procedure Name as a Label

All "statement-number" labels are erased by compilation of the "end" statement. Thus it is not possible to transfer flow into or out of a procedure body by means of such labels. Normal entry to the procedure is through the "enter" statement, and normal exit is from the "exit" statement.

Compilation of the "enter" statement defines the name of the procedure as a label. In some cases it may also be used as a switch. It must not be used

before it is so defined. Thus, if procedure A calls upon procedure B, then B must be compiled before A.

A direct but not obvious result of the restrictions above is that all procedures used by a program must be compiled before any executable statement of the main program. (The list of non-executable statements, which becomes appropriate at this point, is "dim", "index", "dbind".) Another is the exclusion of recursive schemes, in which a procedure may call upon itself, either directly or through a chain (however long) of other procedures.

F. The "Name Block"

The locations immediately before the "enter" statement may be used by or in connection with a procedure in several ways. A block of locations may be reserved by a single statement preceding the "enter" statement consisting of repetitions of the two expressions "stop" and "use'0' ". These may be referred to, either within the procedure or from the calling program, by the name of the procedure with an integer subscript. The subscripts, starting at 1, refer to the locations in the "name block" in reverse order, starting at the end of the statement. If a location contains "use'0' ", the correspondingly subscripted name is a switch. If a location contains "stop", the correspondingly subscripted name is a variable.

A location in the name block may contain "use's4' ", where s4 is used here as an example to denote the label of any statement within the procedure body. In this case, the correspondingly subscripted name is a label; however, it may be used also as a switch, but if it is so used, it becomes a switch during the subsequent flow. In a similar way, if a value is stored in a "switch" or "label" element of the name block, it becomes a "variable" element during the

subsequent flow of the program.

Name block variables are useful for conveying single words of data from the main program to the procedure and vice versa. This is usually more efficient than an entry in the parameter list for a single word.

Name block switches may be used for alternate exits and entries to the procedure. The first entry must always be through the "call" and "enter" system.

Many such uses may be proposed; in connection with them, it should be stated here that the name of a procedure (without a subscript), when used as a label, refers to the normal exit from the procedure (following the "call'name' " statement). The statement "exit" used within the procedure "name" is equivalent to the expression "use'name' ". We also note that the statement

call'name'2''

(for example) is equivalent to the statement

ret'name'2'use'name'1''.

We now present a powerful use of the name block. We suppose that name'1' and name'2' are name block switches. Then the statement

call'name'1''

within the procedure body transfers to the statement following the "call'name'" statement in the calling program, at the same time setting name'1' for a return into the procedure. If now the flow in the calling program encounters the statement

call'name'2''

the flow passes back into the procedure at the statement following the "call'name'1''' statement and name '2' is set up as the exit to the statement following this last statement in the calling program.

Thus the block of coding in the flow of the calling program between "call'name' " and "call'name'2' " becomes a subprogram which may be called at will from within the procedure body by "call'name'1' ".

The extension of this device to several such "slave blocks" is obvious.

A numerical integration procedure, for example, would use the "slave block" to obtain the ordinate at each required abscissa. After the process is completed, execution of the statement

use'name'2''

transfers to the statement following "call'name'2'' in the calling program.

G. "Global" and "Local" Symbols

Normally, all labels and variables used within a procedure body are "local", in the sense that they become undefined and available for reuse after compilation of the "end" statement.

If it is desired to refer to some symbols from the calling program also, this can be arranged. Such nonlocal symbols will be called "global" symbols. "Statement-number" type labels cannot themselves be global but a name block label may be used to accomplish the same purpose. Global variables may be defined by a nonexecutable statement preceding the "enter" statement. Thus,

dim'a'1''

before the "enter" statement makes "a" a global variable.

Formal parameters are usually local. They may be made global by the use of the operator "local". Compilation of this operator makes all symbols preceding it global.

Thus, if we write

```
enter'name'a'b''
local'c'd''
```

then "a" and "b" are global symbols and "c" and "d" are local symbols.

Note that "local" must start a new statement, and that symbols (if any) following it are a continuation of the formal parameter list.

The use of global symbols in procedures is not recommended because of the possibility of conflict between procedures. We have adopted the rule that such symbols, if they are considered necessary, shall consist of five characters commencing with a numeric digit and ending with the first four (or all) characters of the name (separated by spaces if necessary to make five characters).

H. Subscripting Formal Parameters

Formal parameters, as used in the procedure body, must always be subscripted as described in Chapter IX. Thus, if the formal parameter "a" corresponds to a single variable or label (rather than a region) it must be referred to as a'0'; the use of a' (unsubscripted) is incorrect. Combinations such as a'i'2' are not allowed if "a" is a formal parameter.

J. A Procedure Call as an Operand

If a procedure has a single value as its result (or one of its results), this value may be left in the accumulator at exit, either as the result of the last substitution statement executed before "exit" or by replacing "exit" by

```
if'E'exit''
```

where "E" represents the expression whose value is the result. In this case, the "call" with the actual parameter list must be used as an expression of precedence zero in the calling program.

In a similar fashion, if a procedure requires a single value as data, this may be placed in the accumulator before the procedure is called, either as the result of the previous statement or with the operator "if". Thus, the calling sequence for the polynomial evaluation procedure in Appendix D is

```
if'FE'call'polyn'arg'R';'FV'.
```

Here, "FE" represents the floating-point expression whose value is the argument of the polynomial, the coefficients of the polynomial are in region "R", and the result is stored in location "FV". This entrance argument in the accumulator is used by the operator "prev" (see Chapter XI, Section C) within the procedure body.

K. Avoiding Recompilation

Any frequently used group of procedures may be compiled without a main program and punched out in a form that makes recompilation unnecessary. This can save valuable time. Details will be found in the Operator's Manual.

L. Sample Procedures

A study of the procedures described in Appendix D is recommended at this point.

Formal parameters are usually local. They may be made global by the use of the operator "local". Compilation of this operator makes all symbols preceding it global.

Thus, if we write

```
enter'name'a'b''
local'c'd''
```

then "a" and "b" are global symbols and "c" and "d" are local symbols.

Note that "local" must start a new statement, and that symbols (if any) following it are a continuation of the formal parameter list.

The use of global symbols in procedures is not recommended because of the possibility of conflict between procedures. We have adopted the rule that such symbols, if they are considered necessary, shall consist of five characters commencing with a numeric digit and ending with the first four (or all) characters of the name (separated by spaces if necessary to make five characters).

H. Subscripting Formal Parameters

Formal parameters, as used in the procedure body, must always be subscripted as described in Chapter IX. Thus, if the formal parameter "a" corresponds to a single variable or label (rather than a region) it must be referred to as a'0'; the use of a' (unsubscripted) is incorrect. Combinations such as a'i'2' are not allowed if "a" is a formal parameter.

J. A Procedure Call as an Operand

If a procedure has a single value as its result (or one of its results), this value may be left in the accumulator at exit, either as the result of the last substitution statement executed before "exit" or by replacing "exit" by

```
if'E'exit''
```

where "E" represents the expression whose value is the result. In this case, the "call" with the actual parameter list must be used as an expression of precedence zero in the calling program.

In a similar fashion, if a procedure requires a single value as data, this may be placed in the accumulator before the procedure is called, either as the result of the previous statement or with the operator "if". Thus, the calling sequence for the polynomial evaluation procedure in Appendix D is

```
if'FE'call'polyn'arg'R';'FV''.
```

Here, "FE" represents the floating-point expression whose value is the argument of the polynomial, the coefficients of the polynomial are in region "R", and the result is stored in location "FV". This entrance argument in the accumulator is used by the operator "prev" (see Chapter XI, Section C) within the procedure body.

K. Avoiding Recompilation

Any frequently used group of procedures may be compiled without a main program and punched out in a form that makes recompilation unnecessary. This can save valuable time. Details will be found in the Operator's Manual.

L. Sample Procedures

A study of the procedures described in Appendix D is recommended at this point.

Operator	Type	Precedence	Left Operand	Right Operand
Call	F or I	0	None	L
arg	F or I	0	*	V or L
local	N			
enter	N			
end	N			
exit	L		None	None

*left operand must begin with "call"

CHAPTER XI

ADDITIONAL TOPICS

This chapter includes a discussion of several topics of interest primarily to users already familiar with the LGP-30.

A. Machine Language Coding

The ACT III source language includes most of the LGP-30 machine language operation codes. The following table gives the proper forms.

b	if (or "bring")
y	stadd
r	ret
d	div
n	nmult
m	mult
e	extrt
u	use
t	neg (or "trn")
a	add
s	subtr

Where two forms are given, they are interchangeable, but the first compiles more rapidly. Some of these have already been discussed; the remaining ones are all precedence zero, type X, and may have any suitable expression as right operand.

The only missing codes are p, i, and z. The operator "daprt" codes Pxx00, Z0001 for each character, and the operator "rdhex" codes P0000, I0000.

These are the most common uses of these codes.

B. Internal Data Format

Data may be coded for "rdhex" by hand, in the proper format. This information may also be useful in console debugging.

B. 1. Integer Format

All integers are carried at a q of 29. The bit at 30 is zero.

B. 2. Floating Point Format

The fraction is carried in bits 0-25 at a q of 0, rounded to bit 25. The exponent plus 32 is in 26-30 at a q of 30. Floating point zero is carried as a machine zero.

B. 3. Alphameric Format ("aprt").

Words printed by "aprt" are treated as five 6-bit characters. The sign bit must be zero (positive).

C. Previous Result ("prev")

It is frequently possible to increase the run-time efficiency of a program by use of the operator "prev". This is a precedence 3 operator which has no operands, and has the type of the result of the previous statement. To work properly, it must be the first operator compiled in the statement.* If this condition is satisfied, and the previous result was stored in a variable-subscripted location, (or any subscripted dummy parameter location in a procedure), "prev" is always more efficient than calling the value back from the subscripted location. If the

*Note, however, that certain operators, such as "cr", "tab", "neg", "hold", "stadd", do not change the accumulator.

previous result was stored in a nonsubscripted location, "prev" is more efficient unless it is the right operand of a binary operator (one with both left and right operands). Thus, the statement

```
a'-'prev';'b''
```

is correct, but it is more efficient only if the previous result was stored in a variable-subscripted location. However, the statement

```
a'-'b'-'prev':'c''
```

is incorrect, since "prev" is not the first operator compiled.

"Prev" may also be useful in entering a procedure or in any statement which may be reached by several different paths.

D. Direct Address Modification

Statement numbers may be used as operators in direct (machine-language) address modification techniques, provided that the numbered statement so handled is not trace-compiled. A "go to 's0'' statement is never trace-compiled (this is the only reason for its existence as distinct from the "use" operator). The only other situation in which such techniques are safe is within a procedure body.

Initial addresses may be set up with the help of the following information. If "a" is a dummy parameter of the procedure, the location referenced by "a" (unsubscripted) contains the complement of the address of word zero of the actual region (a'0').

Machine language coding may be inserted in the "name block" of a procedure for purposes of storing test constants for address modification loops, or as part of such loops.

E. Overflow and Breakpoint Provisions

Machine overflow cannot occur in the floating point subroutines of the standard system. However, the integer operators "i+" and "i-" use machine operations, and overflow may occur in these or in the operations "add", "subtr", and "div". In the case of a standard logic board this will cause a stop. On the overflow and breakpoint modified logic board, however, no stop will occur, but the overflow indicator is set "on". This may be tested by the program. The statement

oflow'use's7''

will transfer control to statement 7 if the overflow indicator is on, to the following statement (in sequence) if it is off. In the standard system, the overflow indicator is turned off only by pressing "clear counter" or execution of the above statement. If the operating subroutines have been assembled using a modified machine, the overflow indicator is turned off (if it is on) by the floating point subroutines "+" and "-". An error stop will occur if the indicator is on at entry to the "/" subroutine.

The statement

bkp4'use's7''

when run on an overflow logic machine, will transfer control to statement 7 if breakpoint 4 is on, to the next statement if it is off. On a standard machine, this statement does nothing (control to next statement) if breakpoint 4 is on, and causes a stop if it is off. In the latter case, pressing "start" continues with the next statement, while execution of the sequence "one operation, manual input, start, one operation, normal, start'' will transfer to statement 7.

The operators "bkp8", "bkp16", "bkp32" perform similar functions with respect to the corresponding breakpoint switches.

The label "s 7" is used above as a sample label; variable subscripts must not be used on these labels.

F. Efficiency of Object Program

It is not usually important to consider in detail the problem of efficiency; however, in a program which is very tight for storage space or where running time is a vital consideration these problems become important. Thus, for example, of the two statements

```
s1'    a'+'b'x'c';'d''
s2'    b'x'c'+'a';'d''
```

the second is more efficient. This is because the compiler does not recognize the commutative property, and is set up for maximum efficiency when binary operations are performed from left to right. Specifically,
statement 1 above will code two more instructions into the object program (involving two extra nonoptimum storage accesses) as compared with statement 2. In a similar way, the expression

```
a'/'['b'x'c']'
```

is not as efficient as the expression

```
a'/'b'/'c'.
```

As a general rule, greatest efficiency is obtained if, as far as possible, the operators are arranged so that the natural order of execution proceeds from left to right.

The statement

```
i'i+'1';'i'until'm'neg's1''
```

is more efficient than the statement

```
for'i'step'1'until'n'rpeat's1''
```

by three instructions, three nonoptimum storage accesses and one multiplication. It is necessary that m = n+1 in order that these two statements be equivalent. Also, the increment must be positive; if it is negative, simply replace the "neg" by "pos".

Another efficiency rule is that any operator with a result (type I or F) should either be the last operator executed in the statement or the lowest ranking operator of an expression which is used as an operand. Thus, for example, the statement

```
a';'b'use's1''
```

although correct, will compile one unnecessary instruction involving a storage access. This should, therefore, be separated into two statements for greatest efficiency.

The statement

```
n'reprt'x''
```

compiles 10 instructions; it is, therefore, more efficient to use "daprt" if "n" is five or less, since each character to be printed by "daprt" compiles two instructions (a P and a Z). "Aprt" is more efficient than either, if the inconvenience of coding is not considered; each use of "aprt" compiles three instructions and can print up to five characters.

Each variable named in "index" or "dbind" statements causes the compilation of five instructions. Thus there may be some advantage in multiple use of subscripts to as great an extent as the program allows.

While the use of mnemonic names for variables is handy, each such name uses an extra data word. Thus, when overlapping does not occur, double use of such locations may be helpful. If two distinct names are desired (say "pay" and "rate"), they may be assigned a common data location using "dim" as follows.

```
dim'pay'0'rate'1''
```

CHAPTER XII

THE STANDARD SYSTEM

The ACT III system is designed to be extremely flexible. It is possible to change both its vocabulary and syntax in many ways. Methods of doing this are described in Volume 3, the "Technical Manual". For purposes of standardization, however, a "standard system" has been developed. The present volume describes the vocabulary and syntax of this "standard system". It has proven adequate for the great majority of applications. It contains all the features described in this volume and in Volume 2, the "Operator's Manual".

If your program and/or data storage requirements are too large, it is possible to gain extra space by sacrificing some of the operators. If you wish to add to, or modify, the vocabulary this may be done. For details, consult Volume 3.

APPENDIX A

List of Abbreviations

F	floating-point type
I	integer type
L	logic type (operator)
X	operator without result
L	label
LN	label (must not have a variable subscript; in a procedure, must not be a dummy parameter)
S	switch
V	Variable
E	expression
A	alphanumeric type
C	character

APPENDIX B

Table of Characters and Controls

The following page contains a list of all the characters and controls on the LGP-30 Flexowriter, together with the proper two-digit code for "aread" and the proper code for "daprt".

Thus, for example, if the data word 105f0846' is read in by "aread", then the effect of "aprt" using this word will be the printing of upper case, T, lower case, o(that is, the word "To").

The codes 50 and 48 are used only for controlling the Flexowriter when listing tape produced by the high-speed punch. They stop the listing without printing anything. "50" is ignored if the conditional stop button is down.

Lower case	Upper case	"aread" codes	"daprt" codes
a	A	72	a'
b	B	0f	b'
c	C	6f	c'
d	D	2f	d'
e	E	4f	e'
f	F	54	f'
g	G	5j	g'
h	H	62	h'
i	I	22	i'
j	J	64	j'
k	K	6j	k'
l	L	0j	l'
m	M	3f	m'
n	N	32	n'
o	O	46	o'
p	P	42	p'
q	Q	74	q'
r	R	1f	r'
s	S	7f	s'
t	T	5f	t'
u	U	52	u'
v	V	3a	v'
w	W	7j	w'
x	X	4a	x'
y	Y	12	y'
z	Z	02	z'

** Non-printing, HSP only

Lower case	Upper case	"aread" codes	"daprt" codes
0	)	04	0'
1	L	0j	1'
2	*	14	2'
3	"	1j	3'
4	Δ	24	4'
5	%	2j	5'
6	$	34	6'
7	π	3j	7'
8	Σ	44	8'
9	(	4j	9'
Space		06	'
-	_	0a	-'
+	=	16	+'
;	:	1a	;'
/	?	26	/'
.	]	2a	.'
,	[	36	,'

Controls

Lower case	08	lcl'
Upper case	10	uc2'
Color shift	18	color'
Carr. Ret.	20	cr4'
Backspace	28	bs5'
Cond. stop (')	40	stop'
Tabulate	30	tab6'
Cond. stop**	50	stopf'
Uncond. stop**	48	stop9'

APPENDIX C

Table of Operators

The following table lists all the operators included in the standard system.

At this point, it is worth pointing out that this list provides a guide to the words not available for use as variables. The following must be added to the list of forbidden variable names: any 1-5 numeric digits; period (.) or "s" or "+" followed by 1-4 numeric digits.

Also note the special variable "remdr" which is discussed in V-B and should not be used for any other purpose.

Operator	Precedence	Type	Left Operand	Right Operand	Reference (Ch.-Sec.)
;	0	F or I	E	V	V-A;V-B
+	1	F	FE	FE	V-A
-	1	F	FE	FE	V-A
x	2	F	FE	FE	V-A
/	2	F	FE	FE	V-A
abs	3	F	-	FE	V-A
rdflo	0	F	IE	FV	VII-B
read	0	F	-	FV	VII-B
print	0	X	IE	FE	VII-C
dprt	0	X	IE	FE	VII-C
0-	3	F	-	FE	V-A
i+	1	I	IE	IE	V-B
i-	1	I	IE	IE	V-B
ix	2	I	IE	IE	V-B
i/	2	I	IE	IE	V-B
nx	2	I	IE	IE	V-B
use	0	L	-	L	VI-D, IX-G, XI-E

Operator	Precedence	Type	Left Operand	Right Operand	Reference (Ch.-Sec.)
step	0	I	IV	IE	VI-G
for	0	X	-	-	VI-G
until	0	X	IE	IE	VI-G
rpeat	0	L	-	L	VI-G
iread	0	I	-	IV	VII-B
iprt	0	X	IE	IE	VII-C
iabs	3	I	-	IE	V-B
punch	0	X	-	FE	VII-E
ipch	0	X	-	IE	VII-E
aread	0	A	-	AV	VII-D
aprt	0	X	-	AV	VII-D
if	0	X	-	E	VI-E, XI-A
neg	0	L	-	L	VI-E , XI-A
zero	0	L	-	LN	VI-E
pos	0	L	-	L	VI-E
prev	3	F or I	-	-	XI-C
cr	0	X	-	-	VII-C
exit	-	-	-	-	X-C
local	-	-	-	-	X-G
end	-	-	-	-	X-C
trace	-	-	-	-	Vol. 2,
wait	-	-	-	-	VI-H
hxpch	0	X	-	E	VII-E
bring	0	X	-	E	XI-A
add	0	X	-	E	XI-A
subtr	0	X	-	E	XI-A
mult	0	X	-	E	XI-A
nmult	0	X	-	E	XI-A
div	0	X	-	E	XI-A
extrt	0	X	-	E	XI-A
hold	0	X	-	V or L	XI-A
clear	0	X	-	V or L	XI-A

Operator	Precedence	Type	Left Operand	Right Operand	Reference (Ch.-Sec.)
stadd	0	X	-	V or L	XI-A
tab	0	X	-	-	VII-C
flo	3	F	IE	IE	V-C
unflo	3	I	IE	FE	V-C
fix	3	I	IE	FE	V-C
x10p	3	F	FE	IE	V-A
index	-	-	-	IV	IX-C
dbind	-	-	-	IV	IX-E
dim	-	-	-	-	IX-A
daprt	0	X	-	-	VII-D
reprt	0	X	IE	C	VII-D
stop	0	X	-	-	VI-H
call	0	F or I	-	LN	X-C
arg	0	F or I	E	V	X-C
ret	0	L	-	S	VI-F
trn	0	L	-	L	XI-A
rdxit	0	L	-	LN	VII-B
sqrt	3	F	-	FE	V-A
ln	3	F	-	FE	V-A
log	3	F	-	FE	V-A
exp	3	F	-	FE	V-A
pwr	3	F	FE	FE	V-A
ipwr	3	I	IE	IE	V-B
sin	3	F	-	FE	V-A
cos	3	F	-	FE	V-A
artan	3	F	-	FE	V-A
set	0	L	-	S	VI-F
to	0	L	-	LN	VI-F
go to's0'	-	X	-	-	VI-F
bkp4	0	L	-	-	XI-E
bkp8	0	L	-	-	XI-E
bkp16	0	L	-	-	XI-E

Operator	Precedence	Type	Left Operand	Right Operand	Reference (Ch.-Sec.)
bkp32	0	L	-	-	XI-E
oflow	0	L	-	-	XI-E
randm	3	F	-	-	V-A
rdhex	0	F, I, A	-	V	VII-E
enter	-	-	-	-	X-C

APPENDIX D

Sample Programs

This appendix contains sample programs designed to illustrate programming techniques.

Each complete program is documented to illustrate the use of the system. Specifically, this documentation includes the following: source program, storage map (see Vol. 2), sample data, and results of sample run.

Three procedures are also included; two of these, POLYN and PLYDR, are used in Sample Program No. 5. The third, MOVE, is included to demonstrate how the procedure provisions make it possible for sophisticated routines to be utilized by the average user. This procedure makes extensive use of LGP-30 hardware features and depends upon a detailed knowledge of the inner workings of the compiler, but it may just as easily be used by persons lacking this knowledge.

Notes on Labeling Data

The data tapes listed in this appendix illustrate the simplest technique for including labels and comments on data tapes. We will summarize some rules mentioned in the manual, and add a few more comments.

First, any remark ending with eight spaces may precede a positive integer on the data tape; in this case, the "+" sign must be omitted. If a remark is placed before a negative integer, the integer must be written with a full seven digits (using leading zeros as necessary).

A remark may precede a floating point datum; in this case, the mantissa must be written with a full seven significant digits (the decimal point is understood to follow the sign).

Another useful technique is to include a "rdhex" in the program, storing into an unused location. This will then read any string of characters and controls ending with a stop code (or, if you do not want the stop code printed, roll the tape back one character after punching it and then overpunch with "upper case").

Sample program no. 1

Accompanying Chapter VI section G, to demonstrate a "for" statement.

This program reads an integer "n", then reads n pairs of floating point numbers "a" and "b" and prints them together with their sum, difference, product and quotient. A test prevents division by zero. A blank word for "n" causes a stop in sl0.

```
HJB : Programmer'

     rdxit'sl0''
sl'  iread'n''
     1';'i''                initializes loop:remark'
     if'n'zero'sl''          tests to see if loop should be executed at all:remark'
s2'  read'a''
     read'b''
     cr'1305'print'a''
     1305'print'b''
     1305'print'a'+'b''
     1305'print'a'-'b''
     1305'print'a'x'b''
     if'b'zero's3''        to prevent division by zero:remark'
     1305'print'a'/'b''
s3'  for'i'step'1'until'n'rpeat's2''
     cr'use'sl''           return to read new "n":remark'

sl0' stop''                 here on blank "n":remark'
     use'sl'''

f 0463

s001  0307
s002  0322
s003  0442
s010  0457

remdr  4336
    n  3063
    i  3062
    a  3061
    b  3060
```

Sample program no. 1

Data -- to be separated from output

```
       2'
+1'+1'+1'+1'
+2'+1'+1'+1'

+3'
+1'+1'+0'+0'
-5'+1'+2'-3'
+1'+1'-1'+1'

'
```

```
  .10000 e 01  .10000 e 01  .20000 e 01  .00000 e 00  .10000 e 01  .10000 e 01
  .20000 e 01  .10000 e 01  .30000 e 01  .10000 e 01  .20000 e 01  .20000 e 01

  .10000 e 01  .00000 e 00  .10000 e 01  .10000 e 01  .00000 e 00
 -.50000 e 01  .20000 e-03 -.49998 e 01 -.50002 e 01 -.10000 e-02 -.25000 e 05
  .10000 e 01 -.10000 e 01  .00000 e 00  .20000 e 01 -.10000 e 01 -.10000 e 01
```

Sample program no.2

Accompanying Chapter VIII section A.

Compute mean and standard deviation.

This program reads a set of numbers "y", and computes and prints their mean and standard deviation.
The data tape contains, first, the run number, followed by the integer "decy" giving the decimal point position for the "y" values. These are then read in by decy*rdflo*y**. A blank "y" signals the end of the set. A blank "run" stops the operation in s10.

HJB: Programmer'

```
s1'  rdxit's10''
     iread'run''
     iread'decy''
     rdxit's5''
     0';'Σy';'Σysq';'n''      initializing counter and sums:remark'
s3'  decy'rdflo'y''
     y'+'Σy';'Σy''
     y'x'y'+'Σysq';'Σysq''
     n'i+'1';'n''
     use's3''
s5'  0'flo'n';'en''        here at end of set:remark'
     daprt'cr4'cr4'uc2'R'lcl'u'n' 'n'o'.''
     0'iprt'run''
     cr'0'iprt'n'daprt' 'c'a's'e's''
     if'n'zero's1''      to prevent division by zero:remark'
     Σy'/'en';'ybar''
     sqrt'['Σysq'/'en'-'ybar'x'ybar']';'sigma''
     daprt'cr4'uc2' 'lcl'cr4'y' 'uc2'='lcl''
     1600'print'ybar''
     daprt'cr4's'i'g'm'a' 'uc2'='lcl''
     1606'print'sigma''
     use's1''
s10' stop'use's1'''
```

Sample program no.2 -- continued

```
f 0634

s001  0302
s003  0328
s005  0401
s010  0630

remdr  4336
  run  3063
 decy  3062
   8y  3061
 8ysq  3060
    n  3059
    y  3058
   en  3057
 ybar  3056
sigma  3055
```

This trial was run with both input and output through the Flexowriter. The result is given below.

```
Sample program no. 2

Test data

Run no.        1'
+2'+100'+200'+300'+155'+90'+48'+401'+253'+322'+298''

Run no. 1
 10 cases

ȳ =  .21669984 e 01
sigma =  .31051631 e 01

Run no.       2'
+0'+5731'+2985'+3555'+4822'+2500'+5052'+3333''

Run no. 2
 7 cases

ȳ =  .39968550 e 04
sigma =  .11149549 e 04

'
```

From data tape

From data tape

Output

From data tape

Output

Sample program no.3

Accompanying Chapter VIII section B.

This program demonstrates the use of a "ret-use" block. The block beginning at s5 and ending at s6 computes and prints the sine and cosine of "y". The main program reads two variables "a" and "b" and uses the block successively to treat a, b, a+b, and a-b as "y".

```
HJB : Programmer'

     rdxit's2''
s1'  read'a''
     read'b''
     daprt'cr4'cr4'a' 'uc2'='lc1''
     1608'print'a''
     daprt';' 'b' 'uc2'='lc1''
     1608'print'b''
     daprt'cr4'cr4' 'y' ' ' ' 's'i'n' 'y' ' ' ' ' ' 'c'o's' 'y'cr4'cr4' 'a' ''
     a';'y''
     ret's6'use's5''
     daprt'cr4' 'b' ''
     b';'y''
     ret's6'use's5''
     daprt'cr4'a'+'b'',
     a'+'b';'y''
     ret's6'use's5''
     daprt'cr4'a'-'b''
     a'-'b';'y''
     ret's6'use's5''
     use'sl''

s2'  stop'use'sl''          here on blank word:remark'

s5'  906'dprt'sin'y''
     1006'dprt'cos'y''
s6'  go to's0'''

f 0629

s001  0307
s002  0602
s005  0606
s006  0628

remdr  4336
    a  3063
    b  3062
    y  3061
   s0  0632
```

Sample program no. 3

Test data -- to be separated from output

```
+1000000'+1'+1'+1'
+2'+1'+3'+1'
'
```

```
a =  .99999994 e 00; b =  .99999994 e 00

 y    sin y     cos y

 a   .841471   .540302
 b   .841471   .540302
a+b  .909297  -.416147
a-b  .000000  1.000000

a =  .19999999 e 01; b =  .30000001 e 01

 y    sin y     cos y

 a   .909297  -.416147
 b   .141120  -.989992
a+b -.958925   .283661
a-b -.841471   .540302
```

Sample program no. 4

To illustrate the numeric output format.

The program prints floating point numbers under various format controls to illustrate "print", "dprt", and "iprt-unflo". Each output field is delimited at left and right by an asterisk(*). Note the following points. In "print", provision is made for rounding .999999 up to 1.00 by increasing the exponent. In "dprt", if a number is too large to fit the field, the number of fractional digits is decreased (i.e. the decimal point is shifted to the right within a constant field width). In "iprt", however, the specified number of fractional digits is always printed; the field is extended as required. Also note that, in the case of numbers less than 1.0, "iprt" prints a zero before the decimal point, whereas "dprt" does not. The accuracy of internal representation of floating-point numbers also is demonstrated by comparison of the first output column with the data.

The program reads floating-point numbers as input and prints them in the indicated forms.

```
HJB: Programmer'

     rdxit's5''
s1'  daprt'cr4'cr4'tab6' ' ' ' 'l'6'0'8'stop'p'r'i'n't'stop' ' ' ' 'l'0'0'2'stop''
     daprt'p'r'i'n't'stop' ' ' ' '7'0'3'stop'd'p'r't'stop' ' ' ' '7'0'3'stop''
     daprt'i'p'r't'stop'3'stop'u'n'f'l'o'cr4'cr4''
s2'  read'a''
     daprt'tab6'uc2'*'lcl''
     1608'print'a'daprt'uc2'*' '*'lcl''
     1002'print'a'daprt'uc2'*' '*'lcl''
     703'dprt'a'daprt'uc2'*' '*'lcl''
     703'iprt'3'unflo'a'daprt'uc2'*'lcl'cr4''
     use's2''
s5'  stop'use's1'''

f 0651

s001  0307
s002  0521
s005  0647

remdr  4336
   a   3063
```

Sample program no. 4--continued

The first line below is the data tape, which, because of the design of the program, must not contain comments or carriage returns.

```
+1'+1'-75'-1'+523'+2'+523'+4''
```

The run shown below was made with both input and output on the Flexowriter. The first column contains the input data, the remainder is the computer output.

```
                    1608'print'     1002'print'     703'dprt'     703'iprt'3'unflo
+1'+1'          *  .99999994 e 00* *  .10 e 01* *  1.000* *  1.000*
-75'-1'         * -.75000000 e-01* * -.75 e-01* *  -.075* * -0.075*
+523'+2'        *  .52300000 e 02* *  .52 e 02* * 52.300* * 52.300*
+523'+4'        *  .52300000 e 04* *  .52 e 04* * 5230.0* * 5230.000*
'
```

Sample program no. 5

To illustrate the use of procedures.

This program makes use of two procedures which are listed on succeeding pages. These are POLYN, which evaluates a polynomial, and PLYDR, which differentiates a polynomial.
This program reads first the coefficients of the polynomial into region "p". This region is 100 words long, and provision is made to prevent an attempt to use higher degree polynomials. The procedure PLYDR is used to produce the coefficients of the derivative polynomial in region "d", and the procedure POLYN is then used twice for each value of the argument read in as data, once with "p" and once with "d".

Data: Coefficients of the polynomial (Fl.pt.), starting with the constant term and ended by a blank word; then values of the argument (Fl.pt.), ended by a blank word. The last case is followed by a second blank word.

Output: for each argument value, the value, the polynomial, and the derivative are printed on a single line.

HJB: Programmer'

```
dim'p'101'd'101''
index'i''

s1'  rdxit's3''
     0';'i''
s2'  read'p'i'1''       reads coefficients, constant term first:remark'
     rdxit's5''
     for'i'step'1'until'100'rpeat's2''
s3'  stop'use's1''    here if too many coefficients, or on last case:remark'
s5'  i';'p''           degree of polynomial + 1:remark'
     rdxit's1''
     read'ex''
     cr'if'ex'call'plydr'arg'p'arg'd';'deriv''  first time only:remark'
s6'  if'ex'call'polyn'arg'p';'pol''
     cr'1608'print'ex''
     1608'print'pol''
     1608'print'deriv''
     read'ex''
     if'ex'call'polyn'arg'd';'deriv''
     use's6'''
```

Sample program no.5 -- continued

```
f 0626

s001  0450
s002  0459
s003  0517
s005  0521
s006  0545

remdr  4336
polyn  0302
plydr  0343
    p  3059
    d  2922
    i  0449
   ex  2749
deriv  2748
  pol  2747
```

Sample program no. 5

Test data -- to be separated from output

```
+1000000'+1'-1'+1'+5'+0'+8'-2''
+1'+1'+0'+0'+5'+3'-1'+0'+1'-10'''
```

Program 24.0 HJB (edited 5/26/61)

POLYN procedure'

```
enter'polyn'a''
index'i''

     prev';'ex''
     a'0';'i''
     if'a'i''
s1'  prev';'y''
     i'i-'1';'i'until'1'neg's2''
     if'ex'x'y'+'a'i'use's1''
s2'  if'y'exit''

end''wait'
```

Polynomial evaluation procedure

Calling sequence: if*FE*call*POLYN*arg*V1*:*V2**

This procedure evaluates the nth degree polynomial whose coefficients are in "V1" as a standard row vector (n+1 in V1*0*(integer), coefficient of the ith power of x in V1*i+1*, i = 0....n(floating point)).

If the argument (value of FE) is in the accumulator as the result of a previous operation, the "if*FE*" may be omitted. The floating point answer is stored in V2.

Program 25.1

PLYDR procedure 9/13/61

Uses POLYN '

```
enter'plydr'p'd''
index'i'j''

     prev';'y''
     p'0';'n''
     prev'i-'1';'d'0''
     if'prev'neg's2''
     1';'i''
s1'  i'i+'1';'j''
     0'flo'i'x'p'j';'d'i''
     i'i+'1';'i'until'n'neg's1''
     if'y'call'polyn'arg'd'0''
     exit''
s2'  if'0'exit''

end''wait'
```

Evaluate the first derivative of a polynomial.

Uses POLYN procedure.

Calling sequence: if*FE*call*PLYDR*arg*R1*arg*R2*;*FV**

Here (FE) is the value of the argument of the polynomial, the region R1 contains the polynomial in standard form (degree+1 as integer in R1*0*, coefficient of the ith power of x in R1*i+1*). The derivative polynomial is placed in R2 and the value of this for argument (FE) is placed in FV. R2 and R1 may be the same region; in this case the original polynomial is lost.

MOVE procedure

Calling sequence: bring*n*call*move*arg*a*arg*b**

Moves n words beginning at a*0* to locations beginning at b*0*. If n is in the accumulator as the result of the previous statement, the "bring*n*" may be omitted. If the regions a and b overlap, the moving is arranged so that no information is written over before it is moved.

Running time: about 600 + 56n milliseconds.

Storage requirement: 52 sectors.

Symbols used (global): MOVE, 1move, 2move.

Alternative Call: bring*n*ret*move*use*move*1**

This call may be used if the previous use of the procedure involved the same blocks. It may also be used when this procedure is called from another procedure, in which case the indirect address references from the calling procedure must be placed in "1move" and "2move" before calling "move". Thus, as an example, within the procedure headed by "enter*proc*1proc*2proc*" the block at "1proc" may be moved to "2proc" by the coding

```
1proc*:*1move**
2proc*:*2move**
bring*n*ret*move*use*move*1**.
```

The contents of "1move" and "2move" are not altered by the procedure.

Alternative running time: about 150 + 56n milliseconds.'

```
stop'stop'stop'stop'stop'stop'use's8''
enter'move'1move'2move''
local''
s8'  clear'move'6'subtr'1024'nmult'move'6'use's3'stop''
s6'  prev'i+'move'3';'move'2''
s1'  bring'['s1']'use's2''
s2'  clear'['move'4']'use's5''
s3'  prev';'move'2''
     1move'i-'2move';'move'5''
     prev';'move'7''
     neg's4'bring'4095'clear'move'3''
s7'  subtr'1move''
s5'  add'move'2'stadd's1'add'move'7'stadd's2'subtr'move'5'neg's6'exit''
s4'  bring'4097'clear'move'3''
     prev'i-'move'6'i+'1''
     use's7''
end''wait'
```

GENERAL PRECISION INCORPORATED

ELECTRONIC DATA PROCESSING

C 0011

PRINTED IN U.S.A.