ANL-6697
Mathematics and
Computers
(TID-4500, 24th Ed.)
AEC Research and
Development Report

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60440

# A PRIMER ON THE ACT-III COMPILER FOR THE LGP-30 DIGITAL COMPUTER

by

H. C. Thacher, Jr. and R. E. Grench

Reactor Physics Division

October 1963

Operated by The University of Chicago
under
Contract W-31-109-eng-38
with the
U. S. Atomic Energy Commission

## TABLE OF CONTENTS

## TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

## LIST OF FIGURES

# A PRIMER ON THE ACT-III COMPILER FOR THE LGP-30 DIGITAL COMPUTER

by

H. C. Thacher, Jr. and R. E. Grench

## I. INTRODUCTION

Automatic digital computers offer a means of relieving scientists and engineers of much tedious calculation. Our largest engineering and scientific projects depend upon automatic computers to process and evaluate theoretical or experimental data. Any complicated calculation which must be repeated a dozen or more, or even fewer, times is a worthwhile application for the computer. This is particularly true if the computer is readily accessible without excessive administrative details.

The many ways of using a computer vary in scope and speed. For small problems in particular, it is desirable to lay as much of the burden as possible upon the computer. The difficulty arises in communicating with a programming specialist. The vernacular of the scientist or engineer must be programmed for translation to the language obeyed by the computer. The translation is accomplished by a special program called a compiler. In most cases, therefore, the scales weigh heavily in favor of the scientist or engineer writing his own program.

The purpose of these general notes is to give an introduction to the writing of programs for the ACT-III compiler for the General Precision LGP-30 Computer, and to the procedures for translating and solving problems with them. This compiler was developed by Dr. Henry J. Bowlden of Union Carbide, Cleveland, Ohio. More detailed specifics of the language and of the mode of action of the compiler are available in the manuals for ACT-III, which he prepared. These manuals are distributed through POOL, the LGP-30 Users Organization.

The reader must acquaint himself with the larger portion of this primer to write his own programs. Sections XI, XV, XVI, XVII, and XVIII can be left for reading last, as they contain more advanced programming or are not necessary for the writing of simple programs. The authors strongly recommend an initial scanning of the whole primer and then additional work on the basic portions and example problems.

# II. NUMBERS

Operation of the LGP-30 computer is typical of modern, high-speed, digital computing equipment: calculations are performed primarily by combining a series of numbers through various arithmetic operations. Ordinarily, we are not concerned about the way numbers are handled inside the computer, but we must know how to get them in and how they appear when printed out.

Therefore, we begin by discussing the way in which numbers are written for the ACT-III compiler. First, we must distinguish between two kinds of numbers: integers, and real or floating-point numbers. These are represented differently in the program, are combined by different operations, and are often used for different purposes. Secondly, numbers which enter a program can be of two classes: (1) program constants, which will be the same each time the program is used; and (2) problem parameters, which may vary from case to case.

The programmer can prescribe a variety of formats for output, which will be described at a later stage.

## A. Integers

Integers are used primarily for counting, but they are also valuable in other applications. ACT-III allows the use of positive and negative integers between -536,870,911 and +536,870,911. However, a special multiplication operation must be used if the product exceeds 134,217,727 in magnitude.

### 1. Integer Program Constants

Integer program constants are restricted to positive values. They can be entered in either of two forms:

(1) Up to five digits, followed by a stopcode ('); for example, 123', 0', 12345',

or

(2) A plus (+) sign, followed by one to four digits, a stopcode, and zero to five digits, and another stopcode; for example, +1'23451', +123'456', +123'.

### 2. Integer Problem Parameters

Integers which are problem parameters, or data, may be of either sign, but are limited to a maximum of seven digits. The format consists of a sign and one to seven digits, followed by a stopcode; for example,

+1234567', -14', +0000563'.

If tabs or other characters are used to separate data, the full sign and 7-digit representation must be used.

## B. Real Numbers

Most calculation is done with real, or floating-point numbers.* In ACT-III, real numbers consist of a signed fraction, with magnitude between 0.1 and 1.0, and slightly less than eight decimal-place accuracy, and an exponent between -32 and 31. The value of the number is the product of the fraction and ten to the value of the exponent. In addition, zero can represent either a real number or an integer. Floating-point arithmetic relieves the programmer from estimating the magnitudes of intermediate results, which is otherwise necessary to avoid exceeding the capacity of the machine. It is, however, slower and less accurate.

### 1. Real Program Constants

Real program constants are limited to positive values. They consist of the following components which must be specified in the sequence cited:

(1) decimal point;

(2) one to four digits (the first digit cannot be zero);

(3) stopcode,

(4) zero to five digits;

(5) stopcode,

(6) exponent e (or e- if the exponent is negative);

(7) absolute value of the exponent as a 1- or 2-digit number;

(8) final stopcode

*We will use the terms "real" and "floating-point" interchangeably. The term "real," as used in the international algorithmic language Algol, describes a number which can take on any positive or negative value, or zero. The term "floating-point" describes a particular way (in ACT-III, it is the usual way) of representing a real number in the computer. The floating-point representation is closely related to ordinary scientific notation whereby very large and very small numbers are represented with a scaling factor of a power of 10

Thus, as a real program constant,

$$100{,}000.7 = 0.1000007 \times 10^6$$

would be expressed in the form

.1000'007'e'6'.

Similarly, the constant

$$0.00105 = 0.105 \times 10^{-2}$$

would take the form

.105"e-'2'.

When used for program constants, the floating-point zero and the fixed-point zero are both represented by zero.

Despite the apparent ability to specify up to nine significant digits in the ACT-III compiler, only the first eight digits are used in the computer.

2. Real Problem Parameters

Real problem parameters, or data, are specified and arranged in the following sequence:

(1) plus or minus sign;

(2) one to seven digits (the first digit cannot be zero);

(3) stopcode;

(4) plus or minus sign (for the exponent);

(5) one or two digits (the exponent);

(6) final stopcode.

Thus, as a real problem parameter

100,000.7

would be expressed in the form

-1000007'+6'.

Similarly, the parameter

-0.00105

would take the form

-105'-2',

and zero could be written as

+0'+0'.

## EXERCISES

1. Express the following integers as integer program constants and as integer problem parameters. If it is impossible to do so, indicate why.

| | | | |
|---|---|---|---|
| a.) | +1 | e.) | -536,870,911 |
| b.) | +321456 | f.) | 0 |
| c.) | -52 | g.) | +742,125,000 |
| d.) | +536,870,911 | h.) | +3.1416 |

2. Express the following integer program constants as integer problem parameters and as integers. If it is impossible to do so, or if the "program constant" is incorrect, indicate why.

| | | | |
|---|---|---|---|
| a.) | +0' | e.) | 1' |
| b.) | -1' | f.) | 102" |
| c.) | +1234" | g.) | +7000'00000' |
| d.) | +1'23456' | h.) | -700'0000' |

3. Express the following numbers as floating-point program constants and as floating-point problem parameters. If it is impossible to do so, indicate why, and, if possible, give the nearest approximation.

| | | | |
|---|---|---|---|
| a.) | 0 | e.) | 3.14159265 |
| b.) | 15.0 | f.) | $5.3 \times 10^{31}$ |
| c.) | $6.02 \times 10^{23}$ | g.) | $-0.195 \times 10^{-32}$ |
| d.) | $-3.00 \times 10^{10}$ | h.) | $0.253 \times 10^{-32}$ |

4. Express the following floating-point program constants as numbers and as floating-point problem parameters. If it is impossible to do so, or if the "program constant" is incorrect, indicate why.

a.) .512'34678'e'5'

b.) .5"e-'32'

c.) .7"e'32'

d.) -.4"e'0'

e.) +.512"e'5'

f.) .512"e'-5'

g.) .512'e-'5'

h.) .51234'2678'e'0'

5. Express the following floating-point problem parameters as floating-point program constants and as numbers. If it is impossible to do so for certain cases, or if the "problem parameter" is incorrect, indicate why.

a.) +0'+0'

b.) -1234567'89'e'-5'

c.) -12'-2'

d.) +123456'+7'

e.) +1230000'+7'

f.) +0000123'+7'

g.) +1234567'89'-5'

h.) .1234567'-1'

## III. SIMPLE VARIABLES

In most calculations, the same set of operations is performed with several different sets of numbers. These numbers are substituted for single letters which denote the variables in the basic formula or series of formulas being computed.

In similar fashion, ACT-III allows variables which will be given values either by reading in problem parameters or by calculations performed during the program. However, instead of restricting the names of variables to single letters, ACT-III will accept any combinations of up to five letters, or letters followed by digits or other symbols, and ending with a stopcode. The program does not distinguish between upper and lower case letters; hence, A' and a' would represent the same identifier.

Certain words and combinations of letters, digits, and/or symbols are excluded from use as names of variables, since they represent specific operations in the ACT-III vocabulary (see Appendix C). For example, combinations of one to five digits, or a plus sign (+) followed by one to four digits, are interpreted as program constants. The letter x is reserved to denote multiply. Other combinations beginning with the letter s and followed by one to four digits are reserved for labeling statements.

In identifying variables, it is essential that the names be as descriptive as possible to help in understanding the program. The following are examples of acceptable names for simple variables:

eks', ex', delta', f0', tj-1', fbar1', y'.

## EXERCISE 6

Which of the following represent acceptable names for simple variables? Why are the others unacceptable?

| | | | |
|---|---|---|---|
| a.) | templ' | f.) | sine' |
| b.) | temporary' | g.) | a0' |
| c.) | x' | h.) | a*' (or a2') |
| d.) | root' | i.) | s092' |
| e.) | sin' | j.) | +123' |

## IV. THE ASSIGNMENT OPERATOR

One of the most common types of calculation consists of evaluating a series of formulas and substituting the results of the evaluations into other formulas to calculate the desired quantity. For example, the density of a substance d may be expressed by the formula:

$$\text{Density } d = \text{mass}/\text{volume} \quad . \tag{1}$$

The volume of a sphere is given by

$$\text{Volume} = 4 \times 3.141592 \times (\text{radius})^3/3 \quad . \tag{2}$$

The radius, in turn, is given by

$$\text{Radius} = \text{diameter}/2 \quad . \tag{3}$$

We can calculate the density of a material from the mass of a sphere of given diameter by using Eq. (3) to find the radius; by substituting the radius in Eq. (2) to find the volume; and, finally, by substituting the volume in Eq. (1).

Similar calculations may be specified in ACT-III, although the notation is slightly different. Instead of writing the quantity to be determined at the beginning of the formula, we write it at the end; and instead of interposing an equality symbol, we use the assignment operator :'. This operator may be read as "yields," or "replaces," and is really a more exact expression of what we wish to do than is the equality relation. For example, we may write

```
y'+'delta':'y",
```

indicating that we wish to replace y with y + delta. On the other hand, if we were to write

$$y + \text{delta} = y \quad ,$$

we have an impossible equation, except for delta equals zero.

The assignment operator :' assigns the value of the quantity on the left to the variable on the right. When a new value is assigned to the variable, the previous value is lost. Obviously, the right operand of the assignment operator must be a single variable and not, as for most other operators, a more complex expression.

Assignment operators can be used in succession to assign the same value to several variables. For example, the sequence

```
.1"e'1':'a':'b':'c"
```

will give the floating-point value 1.0 to each of the three variables a, b, and c.

It is *not* permissible to use an expression ending with an assignment operator and variable as the left operand for any other operator; for example,

```
.1"e'1':'a'+'b':'c".
```

The desired result of assigning 1 to a and (a + b) to c may be produced by writing

```
.1"e'1':'a"
```

and

```
a'+'b':'c".
```

A more efficient program would result from replacing the last line with

```
prev'+'b':'c",
```

where the special operand prev' denotes the result of whatever operation was last executed. When used in this manner, prev' should be the first operand encountered in the statement.

## EXERCISE 7

What would be the values of a, b, and c after the following sequence of assignments?

```
0':'a"
1':'b"
2':'c"
a':'templ"
b':'a"
c':'b"
templ':'c"
```

Elementary and tedious as this exercise may seem, it does illustrate an excellent way of understanding a complicated program.

## V. ARITHMETIC OPERATIONS WITH REAL NUMBERS

ACT-III provides the usual arithmetic operators for addition, subtraction, multiplication, and division of constants and variables, as well as several more complicated types of combinations, such as exponentiation and the common elementary functions. These operations are provided both for real (floating-point) operands and for integers. Since the real operands are more useful, we will discuss them first.

### A. Fundamental Operations

The basic operations of arithmetic combine two quantities which are identified as right and left operands, since they appear to the right and left of the operation symbol. For example, in the algebraic expression (a + b), a is the left operand, b is the right operand, and + is the operation symbol.

ACT-III provides for addition, subtraction, multiplication, and division of floating-point operands. Each of these operations is represented by a distinct operator symbol followed by a stopcode:

| | |
|---|---|
| Addition | +' |
| Subtraction | -' |
| Multiplication | x' |
| Division | /' |

Thus the ACT-III expression

```
a'+'b'
```

produces the same result as (a + b) in ordinary algebra.

There is one important distinction between ACT-III and traditional algebraic notations for multiplication of factors. In algebra, the factors are displayed in juxtaposition, and the multiplication symbol is omitted; for example, ab denotes the product of a and b.

By contrast, the multiplication operator must be interposed between the factors to be multiplied in an ACT-III expression. Juxtaposition of factors has a different meaning (see Section XIII: Subscripted Variables).

## B. Precedence of Operations

In writing ACT-III expressions, certain rules and conventions must be observed. They are designed to avoid ambiguity with respect to the combining of terms and/or the sequence in which operations are to be performed.

Simple algebraic sums pose no problem. For example, the ACT-III counterpart of the algebraic sum

$$a + b - c + d$$

would be expressed and evaluated as

```
a'+'b'-'c'+'d'.
```

However, when addition and subtraction are combined with multiplication and division, the order in which the operations are performed becomes important. In algebra, this is taken care of by the general convention that, unless otherwise indicated by brackets or parentheses, all multiplications and divisions are performed before additions and subtractions. Accordingly, the expression

$$a \times b + c \times d$$

is evaluated as

$$(a \times b) + (c \times d).$$

In ACT-III, the desired sequence of operations is assured by assigning a precedence number to each operator symbol, as follows:

| Operator | Symbol | Precedence No. |
|---|---|---|
| Multiplication | x' | 2 |
| Division | /' | 2 |
| Addition | +' | 1 |
| Subtraction | -' | 1 |
| Assignment | :' | 0 |

(see Appendix C for listing of all precedence numbers). In evaluating an expression, operations of highest precedence are performed first, then those of next highest precedence, and so on. If two operations of equal precedence are side by side, the one on the left is performed first. This may be important since in computing with a limited number of significant figures,

(a + b) + c

is not necessarily equal to

a + (b + c).

Another example is the expression

a x b/c x d.

In conventional algebra, this might be interpreted either as

(a x b x d)/c

or as

(a x b)/(c x d).

In ACT-III, the "equal precedence" rule would prevail, and the latter interpretation would be evaluated as

a'x'b'/'c'x'd'.

## C. Special Operations

Among the less-common operations, the ACT-III library provides for exponentiation, sign change, square root, natural logarithm, common logarithm, exponential, sine, cosine, arctangent, and absolute value. Each operation is denoted as follows:

| Operation | Operator |
|---|---|
| Exponentiation | pwr' |
| Sign change | 0-' |
| Square root | sqrt' |
| Natural logarithm | ln' |
| Common logarithm | log' |
| Exponential | exp' |
| Sine | sin' |
| Cosine | cos' |
| Arctangent | artan' |
| Absolute value | abs |

The operator randm' produces a pseudo-random floating-point number between 0 and 1. All of these operations are of precedence 3, i.e., they are performed before multiplications or divisions.

The exponentiation operator pwr' has both a left and a right operand. For example, the expression

a'pwr'b'

produces the quantity $a^b$, where both _a_ and _b_ are floating-point constants or variables.

The other operators listed act on a single quantity, i.e., the variable or constant immediately following. Since the operator -' denotes subtraction, it cannot be used to calculate the negative value of a quantity. Instead, we must use the operator 0-', or the equivalent, but somewhat slower operator 0'-'. To calculate the expression

$$a + \sqrt{b}$$

we write

a'+'sqrt'b'.

Obviously, the operators sqrt', ln', and log' can be applied only to positive operands. The angles for sin' and cos', and the result for artan' are expressed in radians.

## D. Brackets

Parentheses, brackets, and braces are used in algebra to enclose groups of terms whose result is to be treated as a single-number expression.

The same technique is employed in ACT-III; however, only a single form of bracket pair, i.e., [']', is used to delimit the desired groupings. A group of constants, variables, and operators enclosed in a pair of these brackets is treated as a single operand for any immediately preceding or following operator. Thus the largest root of the quadratic equation

$$ax^2 + bx + c = 0$$

would be expressed

['0-'b'+'sqrt'['b'x'b'-'.4"e'1'x'a'x'c']']'/'['.2"e'1'x'a']'.

This expression is also illustrative of the following observations on the necessity of using brackets for grouping terms, and/or alternate methods of achieving the same results. First, all sets of brackets are considered necessary in the mode of expression cited. The outer pair (left of the solidus) is required to specify that

$$-b + \sqrt{b^2 - 4\ ac}$$

is the numerator. If they were omitted, only the square root would be divided by 2a. The inner pair of brackets delimits the discriminant of the square root operator. If they were omitted, the numerator would be interpreted as

$$(-b + b^{3/2} - 4\ ac).$$

The brackets to the right of the solidus define the denominator. If they were omitted, the root would be

$$(-b + \sqrt{b^2 - 4\ ac}) \times a/2.$$

The brackets in the denominator could be eliminated and essentially the same results could be obtained by writing

```
['0'-'b'+'sqrt'['b'x'b'-'.4"e'1'x'a'x'c']']'/'.2"e'1'/'a'.
```

This would be evaluated as

$$([(-b + \sqrt{b^2-4ac})]/2)/a.$$

A second observation, with respect to the basic illustration, is that bracketed expressions may occur inside other brackets. In ACT-III, up to seven sets of brackets may occur in a nest. This is sufficient to meet almost all needs. If more brackets are required, they can be written to some level less than seven along with instructions to assign the result to some temporary variable. Thus

```
a'/'['b'+'['c'x'['d'+'e']']']':'r"
```

with a bracket depth of three, might be replaced by

```
d'+'e':'templ"

templ'x'c':'templ"

templ'+'b':'templ"

a'/'templ':'r"
```

with no brackets at all. The reader may adjudge one pair of brackets in this example as unnecessary. The inclusion of unnecessary bracket pairs will have no effect on the operation of the program, and the cautious programmer will insert brackets wherever there is any possibility of ambiguity.

Every complete expression should have an equal number of opening and closing brackets. This is a common type of error and is checked by the translator program. A simple manual check consists of assigning numbers (from 1 to 7), in ascending and descending order, respectively, to opening and closing brackets as they are encountered in an expression. Accordingly, the final closing bracket should have the number 0. To illustrate, the following expression has been checked by this method:

```
   1         2       3      2       3      2 1 0

0-'['a'+'sqrt'['b'pwr'['c'x'd']'+'exp'['a'/'c']']']'.
```

EXERCISE 8

If a = 0.1 x $10^0$, b = 0.2 x $10^0$, c = 0.8 x $10^0$, and d = 0.4 x $10^0$, give the values of the following expressions:

a.) a'+'b'x'c'

b.) a'/'b'+'c'x'd'

c.) a'x'b'/'c'x'd'

d.) a'/'b'/'c'

e.) a'-'b'x'd'/'c'

## VI. STATEMENTS AND PROGRAMS

### A. Statements

We are now ready to consider the basic segment of an ACT-III program: the statement. A statement may contain up to 63 words. (A word is a variable, a constant, an operation, or a statement number.) Every statement is terminated by a second stopcode. It is unusual, however, and inadvisable to write the full length of a statement, since in checking the program, only results of complete statements are accessible. Therefore, the difficulty of locating an error in a faulty statement increases rapidly with its size.

In most cases, the grammar of the ACT-III language brings a logical end to a statement long before the maximum length is reached. Certain operations such as the assignment operator, the output operators, and a few others, do not have any result, in the sense of a numerical answer which can be used as a left operand for another operator. Two successive variables, or a variable and a constant, have a special meaning (see Section XIII: Subscripted Variables). Thus it is not possible to follow an operator which does not have an answer by any operator which requires a left operand. Ordinarily, therefore, every sequence of assignment operators with right operands ends a statement.

## B. Statement Labels

Statements may be labeled with statement numbers. Although liberal use of statement numbers is good practice it is not necessary to label every statement. When statement labels are used, however, the label must be the first word of the statement. It consists of the letter _s_ followed by an integer between one and 192, and a stopcode. Statement numbers may be assigned in any order; for example,

```
s1'

s105'

s003' (equivalent to s3').
```

Statement labels are useful in several respects. First, they assist in effecting program checkouts. The statement-by-statement print-out of calculated results (trace) includes statement numbers of all labeled statements. This helps the programmer to locate himself in the print-out. Furthermore, stop orders may be compiled so that the number of the statement appears in the oscilloscope This is convenient in determining the reasons for stops.

Second, statement numbers may be used to re-enter a program after an interruption. The translator for ACT-III produces, among other outputs, a list of the locations of the first instruction of every numbered statement. In the case a calculation is interrupted, either because of machine malfunction or because of work of higher priority, the operator may easily start at any numbered statement. Since output and input devices are particularly prone to failure, it is good practice to label all input and output statements, or at least the first of each group.

Finally, statement numbers may be used to direct abnormal changes, jumps, or alterations in the normal flow of calculation from the end of one statement to the beginning of another. Usage of statement numbers for these purposes will be discussed in a later section.

## C. Programs

A program is a series of statements which directs the carrying out of the entire calculation in the desired manner. The end of a program is indicated by an additional stopcode, following the stopcode which ends the last statement. When this stopcode is recognized by the translator, it signals that the translation is completed. The translator proceeds to output information on actual locations for the programmer and then stops. It is not possible to continue translation following the end of a program, although the translator may be reset to translate an entirely new program.

## EXERCISE 9

Write ACT-III statements assigning each of the following values to the variable res':

a.) $([(-0.25\ z + 0.33333333)z - 0.5]\ z + 1)z.$

This is the fastest and most convenient way of evaluating the polynomial

$$-z^4/4 + z^3/3 - z^2/2 + z,$$

which is approximately equal to ln $(1 + z)$ when $z$ is not too large or too near $-1$.

b.) The following continued fraction is a somewhat better approximation to $\ln(1 + z)$:

$$\cfrac{z}{1 + \cfrac{z}{2 + \cfrac{z}{3 + 0.20000000\ z}}}$$

c.) $$z\left(0.1111111 + \cfrac{1.8888889}{(z + 2.4313725 - \cfrac{0.48058439}{z + 1.5686275)}}\right)$$

d.) $$0.11111111\ z + 1.8888889 - \cfrac{4.5925926}{z + 2.6290323 - \cfrac{0.27098508}{z + 1.3709677}}$$

The expressions in b.), c.), and d.) are algebraically equivalent. Compare them with respect to speed and accuracy of computation.

## EXERCISE 10

Write statements for converting between the rectangular representation of a complex number:

$$z = x + iy$$

and the polar representation:

$$z = \rho e^{i\phi} \quad ,$$

and back. Assume $x > 0$. Write expressions for the real and imaginary parts of the sum, difference, product, and quotient of two complex numbers. Choose names for your variables which assist in understanding your notation.

## VII. ELEMENTARY INPUT AND OUTPUT

To be of any value, a program must be able to accept problem parameter data and to communicate the results. ACT-III provides a considerable variety of operations for this purpose. The most frequently used operations will be described in this section. A few specialized input and output operations will be discussed in Section XVI.

### A. Input

#### 1. Integers

The input format for integer problem parameters was described in Section II. Briefly, they are represented as a sign, either + or -, followed by up to seven digits and a stopcode. The instruction iread' with a right operand causes a number in integer format to be read by the reader and assigned to the right operand. Thus with +15' in the reader, the statement

```
iread'n"
```

would cause n to take the integer value +15.

#### 2. Real Numbers

It will be recalled from Section II that the problem parameter for real or floating-point numbers consists of two sections. The first section consists of a sign, followed by from one to seven digits and a stopcode. The second section includes an integer between -32 and +31, and another stopcode. The instruction read' with a right operand causes a number to be read in floating-point format and assigned to the right operand. Thus, with +5'+3' in the reader, the statement:

```
read'eks"
```

would cause the variable eks' to be given the value $+0.5 \times 10^3$.

## B. Output

Output operations are a little more complicated, since it is desirable to specify both the number and the arrangement, or format, in which it is to be printed. The print instructions discussed in this section require an integer as a left operand, to specify the format, and a right operand to specify the number to be printed.

### 1. Format Integer

The format integer is the same for the three output operations described in this section. It specifies first, the total width of the column in which the number is to be printed (including leading spaces), and, secondly, the number of digits to be printed after the decimal point. If $\underline{w}$ is the width of the column, and $\underline{d}$ is the number of digits to be printed after the decimal point, the format number is

$$f = 100\,w + d \quad .$$

For example, the format number 2008 will cause a number with eight digits after the decimal place to be printed in a column twenty spaces wide.

### 2. Real/Floating-point Output

The standard output operation for a floating-point number is

print'.

The statement

f'print'a"

causes the floating-point number $\underline{a}$ to be printed as a fraction and exponent. If the format number is f, the program prints leading spaces as needed, then a space (if the number is positive or minus sign if negative), decimal point, the fractional part of $\underline{a}$, space, e (or e- if the exponent is negative) and a 2-digit exponent.

The sign, decimal point, and exponent require seven spaces; therefore w must be at least seven before any digits of the fraction can be printed. If w is less than seven, only the exponent will be printed. If w is greater than seven, but w - d is not, the number of fraction digits printed will be reduced. Since the number of significant digits carried by ACT-III floating-point arithmetic system is between seven and eight, the value of d should not exceed eight digits. The format number 1608 will give all the information in a minimum space.

## 3. Fixed-point Output of Floating-point Numbers

Although the standard floating-point output is the most generally useful, there are occasions where an unscaled output with a fixed number of decimals is convenient. The operator

```
dprt'
```

fulfills this need. The statement

```
f'dprt'a"
```

causes the floating-point number _a_ to be printed as an ordinary decimal number, with d (from the format number) digits after the decimal point. If the number is too large to be printed in the space allowed, the number of decimal places is reduced. Otherwise the number is printed in floating-point format, and the last decimal place printed is rounded.

## 4. Integer Output

Numbers stored as integers may be printed by the operation

```
iprt'.
```

The statement

```
f'iprt'i"
```

will cause the signed integer _i_ to be printed. The format number _f_ is interpreted as follows: the number of hundreds gives the width of the field in spaces, as for print' and dprt'. However, if the width allowed is insufficient, the entire integer is printed anyway. A decimal point is inserted arbitrarily _d_ digits from the right of the integer, unless d is zero. The use of a _d_ greater than eight digits gives meaningless results.

To illustrate these operations, the following outputs for _a_ would be obtained from the statements cited as columnar heads:

| a | 1606'print'a" | 1606'dprt'a" |
|---|---|---|
| 1.234567 | .123457e 01 | 1.234567 |
| -0.001234567 | -.123457e-02 | -0.001235 |

With _a_ = 1234567, the statement

```
1606'iprt'a"
```

would produce

```
-1.234567.
```

5. Right Operand

Clearly, the right operand of an input operator must be a single variable; it would be meaningless to assign the value read from tape to, say, ['a'+'b']'. The right operands of output operators, however, may be as complex an expression as desired, provided that the whole expression is enclosed in brackets. Thus, the statement

```
1608'print'['['0'-'b'+'sqrt'['b'x'b'-'.4"e'1'x'a'x'c']']'/'['.2"e'1'x'a']']"
```

would compute and print the larger root of the quadratic equation

$$ay^2 + by + c = 0 \quad .$$

6. Carriage Returns and Tabulates

Two additional operators are provided to assist in controlling the format of the computer output. The operator cr' effects a carriage return and advances the paper one line. The operator tab' moves the carriage to the next tab stop specified.

In typing the program tape, it is convenient to know that carriage returns are ignored by the translator. On the normal Flexowriter, tabs are treated as a blank character. On the 4-mode Flexowriter, they are ignored. Thus, carriage returns always can be used to improve readability, and tabs to separate comments. Tabs may also be used to set off data tapes only if the number following comprises a sign and seven digits. Other uses of the tab are to be avoided if the normal Flexowriter is used. In both Flexowriters, spaces are always treated as characters; all other control keys are ignored.

Liberal use of carriage returns and tabs is suggested for they greatly improve the printout for reading, duplicating, and report writing purposes.

EXERCISE 11

What output would be produced by the following program:

```
cr'iread'n"

read'b"

n'iprt'n"
```

```
n'print'b"
n'dprt'b'"
```

with each of the following sets of input:

```
a.)  +0'+1234567'-5'
b.)  +1605'+1234567'-5'
c.)  +802'+1234567'-5'
d.)  +802'+1234567'+0'
e.)  +200'+1234567'+0'
f.)  +202'-1234567'+5'
g.)  +1608'+1234567'+5'
```

## VIII. ELEMENTARY CONTROL OPERATIONS

In most programs, the control of computer operations is inflexible: statements are executed successively in the order in which they are written. This section describes the methods that can be employed in ACT-III to provide a versatile program. More specifically, the operators which can effect unconditional or conditional transfers, or other digressions, form the basic order of operations.

### A. Unconditional Transfers

The operator use', followed by a statement number, causes the statement labeled with that number to be executed. For example, in the case of the quadratic equation

$$ay^2 + by + c = 0$$

the equation solver can be made to compute the roots of any number of equations by writing

```
s5'     cr'read'a"
        read'b"
        read'c"
1608'print'['['sqrt'['b'x'b'-'.4"e'1'x'a'x'c']'-'b']'/'['.2"e'1'x'a']"
        use's5"'.
```

This program will read a set of values for a, b, and c, print the larger root of the equation, and return for another set of parameter values.

### B. Conditional Transfers

The program which we have just written will work provided that, in all cases, $a \neq 0$ and that $b^2 > 4ac$. If $a = 0$, the computer would be instructed to divide a number by zero and would stop. If $b^2 - 4ac$ was negative, the equation has a pair of complex roots.

#### 1. Depending on Last Result

There are two operators which change the flow of a program under certain conditions. The operator trn', followed by a statement number, causes the numbered statement to be taken next if, and only if, the result of the last operator is negative. If it is positive or zero, the following statement is executed. Thus to continue our quadratic equation example, we might revise the program as follows:

```
s5'     cr'read'a"
        read'b"
        read'c"
        b'x'b'-'.4"e'l'x'a'x'c':'discr"
        trn's6"
        cr'1608'print'['['sqrt'discr'-'b']'/'['.2"e'l'x'a']']"
        cr'1608'print'['0'-'sqrt'discr'-'b']'/'['.2"e'l'x'a']']"
        use's5"
s6'     cr'1608'print'['['0'-'b']'/'['.2"e'l'x'a']']"
        1608'print'['sqrt'['0'-'discr']'/'['.2"e'l'x'a']']"
        use's5"'
```

This program will print real roots on separate lines. If a root is complex, it will print the real part, followed by the complex part on the same line.

## 2. Depending on the Sign of an Expression

Provision can be made for three possibilities that the coefficient _a_ vanishes by including an _if'_ statement. The if' statement consists of the operator if' followed by a variable or an expression in brackets. This, in turn, is followed by _neg'_ and a statement number; _zero'_ and a second statement number; and _pos'_ and a third statement number. One, two, or all three possibilities may be included; however, in the latter case, they must be written in the order given.

Accordingly, the if' statement transfers control to the statement whose number follows neg' (if the expression is negative), or to the statement whose number follows zero' (if the expression is zero), or to the statement whose number follows pos' (if the expression is positive). If none of these conditions is met, the program continues on to the next statement. Thus, we may write

```
s1'     if'['y'-'z']'neg's10'pos's20"
s2'     if'w'neg's30'zero's40"
s3'     if'y'pos's50"
s4'     y'+'z'-'w':'v".
```

Under these conditions the statements will be executed in the following sequence:

Statement <u>s10'</u>, if $y < z$

Statement <u>s20'</u>, if $z < y$

Statement <u>s30'</u>, if $y = z$, and $w < 0$.

If $y = z$ and $w = 0$, statement s40' will be executed next. Statement s50' will be executed next only if $y = z$, $w > 0$, and $y > 0$.

The program for solving quadratic equations can be improved further by including an if' statement to test for the vanishing to the coefficient <u>a</u>.

```
s5'     cr'read'a"
        read'b"
        read'c"
        if'a'zero's7"
        .2"e'1'x'a':'denom"
        b'x'b'-'.4"e'1'x'a'x'c':'discr"
        trn's6"
        sqrt'discr':'discr"
        cr'1608'print'['['discr'-'b']'/'denom']"
        cr'1608'print'['['0'-'discr'-'b']'/'denom']"
        use's5"
s6'     sqrt'['0'-'discr']':'discr"
        cr'1608'print'['['0-'b']'/'denom']"
        1608'print'['discr'/'denom']"
        use's5"
s7'     cr'1608'print'['['0-'c']'/'b']"
        use's5"'.
```

In addition to providing for the possibility that $a = 0$, this program has been improved with respect to the denominator. In the original version, the denominator

```
.2"e'1'x'a'
```

would be computed each time it appeared in a print statement, or twice for each pass through the program. By adding the extra statement

```
        .2"e'1'x'a':'denom"
```

we merely have to save and recall the denominator whenever it is needed.

The ACT-III translator produces a program which follows instructions exactly. If shortcuts are to be introduced, the programmer must supply them. This is not a defect, since on many occasions, what appears to be a logical shortcut may be completely wrong.

## C. Transfers from Data Input

There are many times when we want to read in a series of numbers, but do not know, in advance, how many there will be. The input routines for ACT-III are arranged so that when a data word is read, either by an iread' or a read' operation, and has no sign or digits (for example, a blank word), the program transfers to a numbered statement which has been set earlier in the program.

The instruction which effects this transfer is the operator rdxit'. To be effective, it must be executed by the program before the input instruction. Then the transfer will be to the last statement prefixed with the operator rdxit'. For example, the following program is written to read a set of floating-point data from tape and to compute the mean and standard deviation:

```
s100'   rdxit's50"
        0':'n"
        prev':'sum"
        prev':'sumsq"
s75'    read'data"
        prev'+'sum':'sum"
        data'x'data'+'sumsq':'sumsq"
        .1"e'1'+'n':'n"
        use's75"
s50'    cr'sum'/'n':'sum"
        sumsq'/'n':'sumsq"
        1608'print'sum"
        1608'print'['sqrt'['sumsq'-'sum'x'sum']']"
        use's100'"
```

This routine uses the mathematical identity

$$\sigma^2 = \overline{(x - \bar{x})^2} = \overline{x^2} - (\bar{x})^2 \quad ,$$

where the bar denotes averaging. Observe that we must keep track of the number of data, their sum, and the sum of their squares.

The program down to statement s75' is executed once for each set of data. It is, however, a necessary section of the program, since it initializes the calculation. If the rdxit' operator was not set, the program would transfer to whatever place the last user of the machine had designated, with possibly mystifying results. Similarly, if the variables n', sum', and sumsq' were not set to zero, they would very probably have unexpected values. Initialization of his program is an important responsibility of each programmer.

Also, it may be observed that after all the data have been read and the divisions performed, there is no reason to keep the sums. The same variables are, therefore, used to keep the mean values.

Upon entering this program at statement s100', the program first initializes the rdxit' and the variables, and then proceeds to statement s75', where it calls for data in floating-point format. It adds the data read in to sum', and its square to sumsq', adds one to the count of data which have been processed, and returns for more data. This continues until the end of the list of values. After the last value, an extra stopcode causes the program to proceed to statement s50', and compute and print the average and the standard deviation. The program then returns to try another case.

## D. Miscellaneous Control Operations

### 1. Stop

It is frequently convenient to cause the computer to stop, either because some emergency has arisen in the program or because some phase of the calculation has been completed. The operator stop' will bring this about. The next statement can be executed by pressing the START button on either the computer or the Flexowriter.

If the stop statement is numbered and if the statement was not translated for tracing, the statement number will be shown in binary in the instruction register of the computer oscilloscope. This may be useful for determining which of several possible stops has been reached. Rather than worry about reading hexadecimal, it is convenient to use the statement numbers s1' to produce a single step; s5' to produce two steps;

s21' to produce three steps; and s85' to give four steps. Other numbers can be substituted for these to give equivalently easily recognized patterns.

The stop operation is often useful in data input. Suppose that one section of data is expected to be the same for a large number of runs of the problem. To avoid reproducing these data it would be desirable to be able to use the same tape for these data and merely to vary the second section. It is unsafe to attempt to change tapes when the computer is calling for input. A better solution is, after the orders calling for each section of input, to insert a stop, to allow for changing to the next tape of input data.

### 2. Breakpoint Jumps

Occasionally it is convenient for the operator to be able to direct the course of the program. On computers equipped with an overflow logic board, this can be accomplished through the operators <u>bkp4'</u>, <u>bkp8'</u>, <u>bkp16'</u>, and <u>bkp32'</u>. Breakpoint 32 is ordinarily reserved for print delays and should not be used. When one of these operators is encountered, the program either proceeds to the next instruction, if the corresponding button on the console is down, or skips to the instruction following, if the button is up. Thus, the instructions

```
         bkp8'use's2"
         stop"
   s2'   read'a"
```

would cause the program to stop before reading <u>a</u>, if the breakpoint stop 8 button was up; if it was down, the program would bypass the stop.

On computers with a <u>standard</u> logic board, the breakpoint statement is ignored if the button is down; if it is up, the computer stops. A START COMPUTE will transfer to the next statement. If a transfer is desired to the indicated statement, the following buttons must be pushed: ONE OPERATION; MANUAL; START COMPUTE; ONE OPERATION; NORMAL; START COMPUTE.

The ability of the operator to make decisions and changes while the program is running is often convenient. It is wise to use this ability cautiously. There is no record on the output of the computer in which position the breakpoint buttons were set. Since this output is the principal record of the calculation, it is dangerous to use the breakpoints to make changes which will not be clearly reflected by the results. A very useful, and safe, use of the breakpoints is to provide for optional printing of intermediate results, which may be desirable if a calculation does not turn out as expected. A dangerous use of the technique is for selecting one of two methods of computing. It is too easy to neglect to note the position of the breakpoint, so that at a later time it is not possible to tell which calculation was performed. A more satisfactory way of directing the computer to choose one of two or three alternate paths is by inputting a dummy parameter, which is then examined by the program to determine the course of action.

For example, suppose that at some stage of the program we would like to be able to select either Course a' (starting at s100'), Course b' (starting at s125'), Course c' (starting at s150'), or to continue as before. The following program will provide this capability.

```
s1'     bkp4'use's10"
s65'    stop"
        iread'a"
s10'    if'a'neg's100'zero's125"
s150'   . . . . . . . .
```

Then, if breakpoint 4 is up, the program will stop and call for input whenever it comes to statement s1'. The inputs listed below will cause the corresponding Courses to be followed by the program:

| Input | Course Selected |
|---|---|
| -1' | Course a' |
| +0' | Course b' |
| +1' | Course c' |

If the program is to continue as before, breakpoint 4 is in the down position, causing the input call to be skipped the next time through.

## 3. Overflow Skip

In floating-point operations, overflow is unlikely. If a result with exponent greater than 32 is generated, an error indication is printed, and the computer stops. The same thing happens if capacity is exceeded by the ix' or i/' operators.

Overflow can occur in integer addition and subtraction, and in the machine language operations add', subtr', and div'. Computers with the standard logic board stop when this happens. Computers with the overflow logic board continue, but an internal indicator (the sign bit of the command register) is set. The indicator may be tested, and turned off if it is on, by the operator oflow'. This operator causes the following instruction to be skipped if overflow has not occurred, and to be executed if overflow has taken place since the last execution of this operator. For example, the statement

```
            oflow'use's'7"

      s5'   . . . . . . .
```

will cause s7' to be executed in case of overflow, and s5' otherwise.

## EXERCISE 12

Write a program to compute and print a table of secants and cosecants of angles expressed in degrees. The table should be arranged as follows:

| Numbers of degrees | Secant | Cosecant |
|---|---|---|

Include a blank line before every fifth degree (e.g., before 0°, 5°, 10°, . . .).

## EXERCISE 13

We may approximate the definite integral

$$\int_a^b f(x)dx$$

by the sum

$$(b - a)/n[(1/2)f(a) + f[a + (b - a)/n] + f[a + 2(b - a)/n] + \dots$$
$$+ f[a + (n - 1)(b - a)/n] + (1/2)f(b)] \quad ,$$

where n is an integer greater than zero. The approximation becomes better the larger the value of n. Write a program for integrating a function. Assume that there is a section at s100' which assigns the value of f(y) to the variable f, starting with a particular value assigned to the variable y. After computing f, the program is to return to the statement after the statement use's100". Allow a, b, and n to be inserted as problem parameters.

## IX. ESSENTIALS OF A DEFINITIVE PROGRAM

During the course of writing a program, the connotations of the words, abbreviations, or acronyms selected for the operators and the variables may appear perfectly obvious to the author. A few months later, however, they may appear perfectly obscure to the author or, possibly, to another user of the same program. For this reason, it is essential that the author provide an adequate explanation of any program in which a significant amount of effort has been expended.

## A. <u>Names of Variables</u>

The ability to use any combination of up to five characters eases the task of selecting definitive names for variables. The inclusion of a directory is recommended in cases for which it is necessary to abbreviate or to substitute characters which are not available on the Flexowriter keyboard.

## B. <u>Remarks on Program Tapes</u>

It is most convenient to have this directory, as well as other types of explanatory information, included as a part of the program and not filed separately. The ACT-III language provides a means for incorporating such information directly in the program, from which it is immediately available. No operator or variable name in ACT-III can contain more than five letters or other characters. Strings of characters of more than five letters are interpreted as follows: if the sixth character preceding the stopcode is one of the sixteen letters, <u>tidybrazenchumps</u>, the entire string of characters is ignored. If the sixth character is any other character, the string is treated as a blank word. Thus, for example, if we head our program by

Computation of the Zilch Function. John F. Smith Author',

the sixth character before the stopcode is the letter <u>a</u>; therefore the entire section is ignored. On the other hand, in the program section

```
sl'     if'discr'neg's5'     negative discriminant means complex
                                          roots'.
```

the sixth character before the stopcode is a space; therefore, the stopcode is recognized. It is interpreted as the normal end-of-statement signal. If a second stopcode had followed s5', the comment would have been interpreted as a third stopcode, i.e., the end of the program.

In the first example, the word "Author," although superfluous, served to exclude the program heading from recognition during program translation. Other words may be employed for the same purpose; for example, ending descriptions with the word "Remarks." The words "Program'" and "Procedure" are also useful.

The extent to which documentation should be carried out will vary with the contents and objectives of each program. In general, the governing criteria should favor too much, rather than too little, exposition. As a minimum, each program tape should include:

(1) Title of program;

(2) Author's name;

(3) Date;

(4) Input required (identification of each quantity, and whether integer or floating-point, in the order it is called for by the program);

(5) Output produced (identification of each quantity);

(6) Breakpoint options;

(7) Procedures used, with dates. (A copy of the procedures may well be included.)

In addition, it is usually helpful to include a brief description of how the calculation is done, as well as any limitations on the applicability of the program.

Remarks should be inserted in the program itself whenever there is any possibility that another reader might benefit from an explanation of either the need for a particular step or what it accomplishes.

## C. Remarks on Data Tapes

It is also good practice to identify and explain data tapes. In reading either floating-point or fixed-point problem parameters, only the last eight characters (including spaces and, except on the 4-mode Flexowriter, tabs) before the stopcode are examined. If the full eight characters are used by the data (the first of these must be a sign), any desired remark may be prefixed. For example, the following are equivalent:

| | |
|---|---|
| integer i +0000050' | +50' |
| floating-point Em +1500000'+1' | +15'+1' |

Data tapes should have the user's name, the program with which they are to be used, the date, and some form of identification. Identification of input data is often helpful.

## D. Sample Problems

Another very useful item in program documentation is a sample set of input data, labeled with their significance. Such a sample problem not only illustrates the use of the program, but also gives a convenient check that the program has been properly translated and is functioning in the desired manner.

# X. COMPUTER OPERATION

We have now mastered enough of the ACT-III language so that we can write programs to carry out many tedious calculations. It is appropriate, before we continue our study of the language, to describe how to make the computer obey our instructions.

## A. Preparation of Program Tapes

The first step is to convert the handwritten instructions to a form which can be read by the computer. The LGP-30 computer accepts input from the attached typewriter (Flexowriter), or from paper tapes by the reader of the Flexowriter or of the Photoelectric Reader. When tapes are read by the Flexowriter, a copy of the input is produced by the typewriter. Ordinarily, it is faster and more accurate to use tape for all input.

Tapes are prepared by typing the desired information on the Flexowriter (see Fig. 1). Either the Flexowriter attached to the computer or a spare may be used. When the PUNCH lever is down, every time a key is pushed, a corresponding row of up to six holes is punched in the tape.

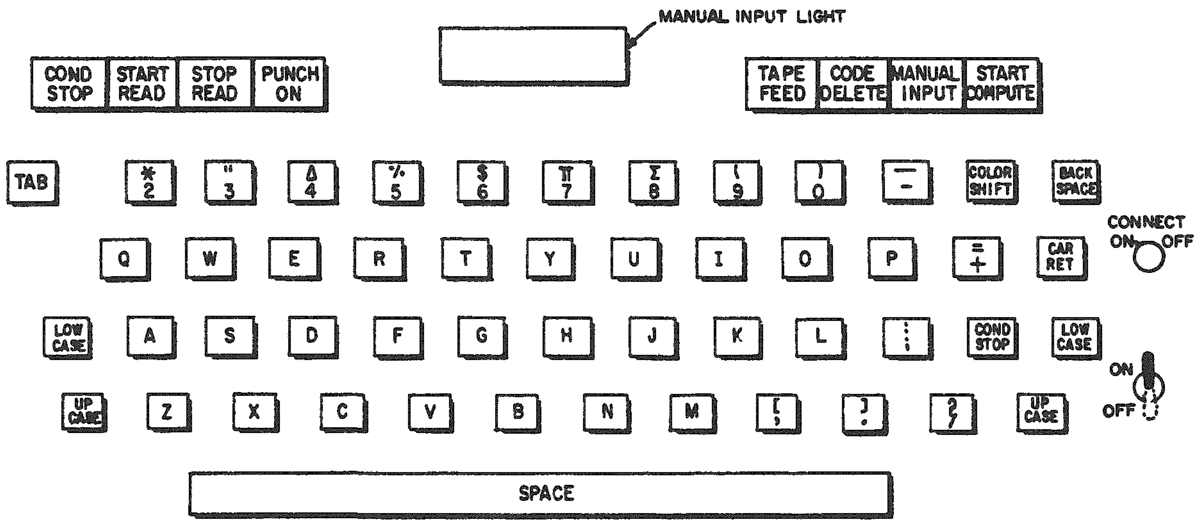

Fig. 1. Flexowriter Keyboard

When the TAPE FEED lever is pressed, the punch feeds blank tape until the lever is released. Every tape should begin with a leader of 10-20 in. of blank tape. This allows space for identifying the tape and facilitates loading the tape in the Photoelectric Reader. Leaders of blank tape are also convenient for separating sections of program or data.

Corrections can be made by either of two methods, depending upon the nature of the errors and the promptness with which they are detected. It is, of course, impossible to erase a set of holes in the tape. However, both readers will ignore any line wherein all six holes are punched. Thus, if detected immediately, an extraneous or an erroneous character may be precluded from translation by simply punching six holes in that particular line. This is accomplished by rolling the tape back one space, and depressing the CODE DELETE lever on the Flexowriter. The same procedure can be followed if several words have been typed incorrectly, provided the errors are detected promptly.

For more serious errors, or those discovered too late, it is more convenient to utilize the ability of the Flexowriter to copy tapes. The procedure is as follows:

(1) Insert the incorrect tape into the reader. Depress the PUNCH and START READ levers. The Flexowriter will then read the tape, and type and punch it, until it comes to a stopcode. When it reaches a stopcode, the reader will print and punch it, and stop.

(2) Depress START READ lever to continue on to the next stopcode.

(3) If it is desired to continue through stopcodes, without stopping, depress the COND STOP lever. Upon nearing the place where corrections are to be made, raise the COND STOP lever, and the reader will stop at the next stopcode. It is also possible to stop the reader by depressing the STOP READ lever.

(4) Type in the correction(s). Then, either roll the incorrect tape forward, or raise the PUNCH lever and allow the reader to read through the incorrect portion of the tape depressing it again when you wish to copy.

## B. Translation

After the program tape has been punched and proofread, it must be translated. The procedure is as follows:

(1) Turn on the computer, the Flexowriter, and the Photoreader (see Figs. 2 and 3). Depress the MANUAL INPUT lever on the Flexowriter. Press the READER STOP button on the Photoreader. Place the ACT-IIIA(S) tape in the Photoreader with the printed side down. Turn the INPUT SELECTOR switch on the Photoreader to READER.

(2) When the warmup cycle of the computer is complete (the oscilloscope shows a pattern), press the ONE OPERATION button, the CLEAR COUNTER button, the NORMAL OPERATION button, and the START button.

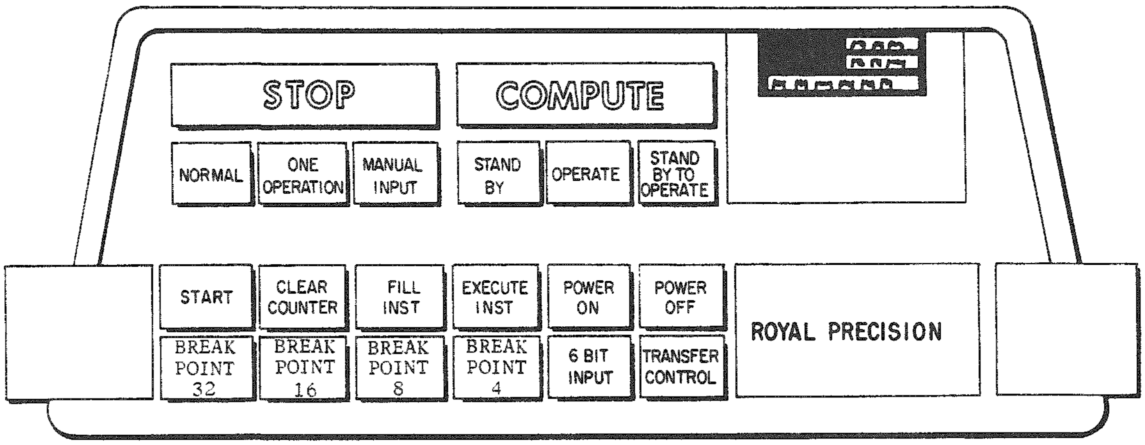

Fig. 2. Computer Control Panel

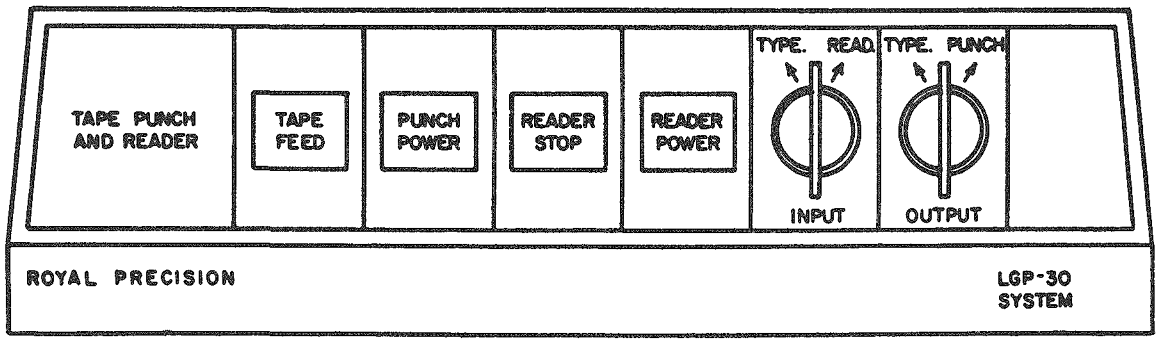

Fig. 3. Photoreader and High-speed Punch Control Panel

(NOTE: Owing to the frequency with which this sequence of operations is performed, it will hereafter be abbreviated to: OCNS.)

All other buttons except OPERATE should be up.

(3) The photoreader will now begin to read the translator tape. While it is being read, the program tape may be placed (with the printed side up) in the reader on the Flexowriter. After the ACT-IIIA(S) tape has been read, the T-tape is placed in the photoreader. (NOTE: In the current edition of the complete compiler, the T-tape is marked T-5.) The ACT-IIIA(S) tape is rewound, and the START button is pressed. After a short interval of computing, the T-tape is read and the computer stops.

(4) Turn the INPUT SELECTOR switch to TYPEWRITER. Depress the 6-BIT button. If a trace is desired, depress the TRANSFER CONTROL button. DO NOT OCNS. Raise the MANUAL INPUT lever. Depress the START button on the computer or, on the Flexowriter, press the START COMPUTE lever.

The program tape will now be read and translated, and the translated program stored in the computer starting at location 0300.

The translator program includes tests for certain common errors in the program, i.e., incorrect operators, unmatched brackets, or exceeding the storage capacity. (A more comprehensive list of errors, as well as the remedy for each, is included in Appendix A.) Whenever an error is detected, the typewriter will print a notification, and the computer will stop. For example, if the program and the data exceed available storage, the Flexowriter will carriage return and print s000 0000. In this case, the program will have to be rewritten. If a statement number is referred to in the program, but is never used as a label, so that it is not defined, the Flexowriter will execute a carriage return and print the undefined statement number. It will also print the location of the machine instruction referring to that statement. This will be repeated for each place where an undefined statement number is used.

When these error indications have been given, or if no errors have been given, the Flexowriter will execute a carriage return, print f, and the machine location of the last instruction of the program. It will then print each statement number used, and the machine address of the first instruction in this statement and, finally, the various variables used, with their machine addresses.

## C. Recompilation

Although it is possible to correct errors in the course of the translation phase, it is better practice to correct the program tape and recompile. In recompiling, or in compiling a second program after a first program has been translated and output, it is not necessary to reload the whole translator and T-tape. Instead, a short tape labeled with T* and the same number as the T-tape being used may be loaded. This resets the program to begin a new translation. Before attempting to load the T*-tape, raise the 6-BIT button, and the TRANSFER CONTROL button. After it is reloaded, return to Step (4) as described above.

## D. Punchout of Object Program

After completing the translation successfully, it is advisable to punch out the translated program. To do this, raise the 6-BIT and the TRANSFER CONTROL buttons, turn on the photoreader, turn the INPUT SELECTOR switch to READER, then press the READER STOP button. Place the ACT-III(B) tape in the reader and OCNS. When the tape has been read completely, turn on the PUNCH on the Flexowriter, feed tape to give an adequate leader, type an identification of the program, and press the START COMPUTE lever on the Flexowriter. The entire program will be punched and printed by the Flexowriter in a form which is not easily readable by the programmer, but is easily reloaded into the computer.

The number of tape-loading operations can be minimized by preparing a single tape which comprises the ACT-IIIA(S), the T-4B, and the ACT-IIIB and C tapes. In this event, the program punch-out procedure is as follows. After translation is completed, depress the MANUAL INPUT lever on the Flexowriter, raise the 6-BIT and the TRANSFER CONTROL buttons, and OCNS. The light on the Flexowriter will illuminate. Type in doat2900', and press the START COMPUTE lever. When the computer stops, turn on the PUNCH, type the program identification, and press the START COMPUTE lever on the Flexowriter.

## E. Running the Program

To run the program, a set of routines to carry out the various operations must be loaded. These routines are contained in a large tape labeled P-4B or P-5B. To load them, turn on the photoreader, raise the 6-BIT and TRANSFER CONTROL buttons and all BREAKPOINTS, and press the READER STOP button. Then place the P-tape in the reader, turn the INPUT SELECTOR switch to READER, and OCNS. If the translated program is still in the memory when the reader stops, turn the INPUT SELECTOR switch to TYPEWRITER, place the data tape in the typewriter reader, depress the MANUAL INPUT lever on the typewriter, and OCNS. When the typewriter light illuminates, type in doat0300', press the START COMPUTE lever twice, and raise the MANUAL INPUT lever. The program will now be executed.

If it is desired to start the program at some other numbered statement, this can be done by finding the true address TTSS (4 digits) from the statement directory printed after translation. Depress the MANUAL INPUT lever, and OCNS. When the light comes on, type doatTTSS' and press the START COMPUTE lever.

If it is necessary to reload the translated program, place it in the photoreader after the P-tape has been read, and press the START button. Then continue by turning the INPUT SELECTOR switch to TYPEWRITER, and so on. However, in this case, OCNS and doat0300' are not necessary after the program is read in.

## F. Checking the Program

The complete checking of a program is difficult and a task requiring skill. The degree of checking will vary with the importance and complexity of the program, and the patience and ingenuity of the programmer. Ideally, every alternative path through the program should be tested to verify that it produces correct results. This can often be done by running the program on several sets of input data with known results.

### 1. Error Indications at Run Time

Theoretically, a carefully checked program should run without interruption and should produce the desired results. This is rarely the case. To the contrary, past experience has shown that many errors in computer fundamentals may be detected painfully rapidly. The majority of these errors can be related to inadvertent instructions to perform illegal operations. For example, dividing by zero, or computing the logarithm or the square root of a negative number.

Whenever an illegal operation is detected, the typewriter will immediately execute a carriage return and type the letter e, followed by a number, and the operator symbol in question. It will then perform a second carriage return and type the number of the last labeled statement executed, the location of the erroneous instruction, and the right operand of the operator. The latter will be interpreted as an integer and as a floating-point number. The types of errors associated with the various operators and the corresponding remedial actions are described in Appendix B.

Even if no illegal operations are detected, the program may still fail to produce the correct result. This, too, indicates an error which must be located and corrected.

### 2. Use of Intermediate Output

In locating errors, it is often possible to get an idea of what may be wrong by studying the output. If the breakpoint options have been used ingeniously to provide extra output of intermediate values, they may be helpful in finding where the program started to go wrong, and what sections are apparently correct. If this device and a careful study of the original program are unsuccessful, tracing may be employed.

### 3. Statement Stopping

It will be recalled that in translating the program, the TRANSFER CONTROL button determined whether the program was to be trace-compiled or not. At run-time when the TRANSFER CONTROL button is up, a trace-compiled program runs the same way as one which is not trace-compiled. If the TRANSFER CONTROL button is down when entering the program, an opportunity is offered for instructing the program to stop at a selected statement number. The operating procedure is as follows: Immediately after entering the program, with the MANUAL INPUT lever down on the Flexowriter and the TRANSFER CONTROL button down, the computer will stop with the Flexowriter light lit. If a + followed by a statement number (without the s) is typed in, the TRANSFER CONTROL button is raised, and the START COMPUTE lever is pressed, the program will run at full speed and stop just before executing the statement specified. Pressing the START COMPUTE lever with the MANUAL INPUT lever down will call for a new statement number. To run without stopping, type in "run." This feature of the ACT-III subroutine system is useful if it is known that the program is all right as far as a certain statement and that only the section after this statement requires examination.

### 4. Tracing

Tracing is a time-consuming task. Moreover, it produces an inordinate amount of output, only a small amount of which is significant. However, tracing does allow the programmer to follow in detail the course of the calculation, to verify each step by hand calculation, and thus to locate his errors.

If the program has been trace-compiled and the TRANSFER CONTROL button is depressed after entering the program, the following print-out will occur for each statement:

Carriage return

Statement number (000, if statement is unnumbered)

Machine address of the first instruction of the statement

Result of statement (interpreted as an integer and as a floating-point number).

## XI. USE OF LIBRARY PROCEDURES AND SUBROUTINES

Procedures and subroutines are blocks of programming which are used repeatedly to perform complicated sets of operations, for example, to evaluate a complex function, to compute the root of an equation, or to invert a matrix at several places in a program. These operations are not provided for directly in the ACT-III language; however, a mechanism is provided whereby procedures from other sources may be incorporated in a particular program. The programmers at most LGP-30 installations maintain a library of procedures for calculations common to their respective organizations, and a number of multiple-use procedures are available through POOL. (The justification of such a library at each LGP-30 installation cannot be overemphasized.)

This section will discuss the method of using procedures which are available. (The rules for writing new procedures are discussed in Section XVII.)

A procedure obtained from a library will, ordinarily, contain special instructions for its use. Among others, these instructions may include call statements to be written in the main program; what the arguments stand for, whether they are integers, floating-point numbers, or sets of numbers; and nature of the results. The discussion here is not intended to supersede these special instructions, but to give a more general description of procedures and their uses.

Multiple-use sections of programming may be classified by the nature of the information which they take in and the information which they produce. The simplest type, for example, is a routine to compute the hyperbolic tangent of y; the input required is limited to the value of y. A more complicated procedure might take a block of data, or several blocks, and produce one or more blocks of data, for example, a procedure which computes the sum of two matrices. Another class of procedures requires a function for one or more inputs. The output might be a single number, or one or more blocks of numbers. Examples are procedures to find a root of an arbitrary function, to integrate a function, or to solve a set of differential equations.

## A. General Call for Procedures

In the ACT-III language, the general call for a procedure consists of a statement, for example,

```
call'beer'arg'blatz'arg'slitz'arg'bud",
```

where beer' is the name of the procedure; blatz', slitz', and bud' are the arguments, which may be single numbers or arrays. One or more of the arguments may be assigned for output or may be changed to a new form by the procedure. The meaning of the arguments and the order in which they are listed will be specified in the description of the procedure.

## B. Special Calls for Procedures

### 1. Functions of One Variable

For procedures in which a single number is required or which produce a single number, a special call can be used to accelerate the computation. This consists of calling the procedure without any arguments, immediately after a statement which leaves the argument in the accumulator. Any expression with a result or an assignment statement will accomplish this.

Subroutines which produce a single number may leave with this number in the accumulator. To assign this result to a variable or to use it in another way, the operator prev' may be used. For example, the sequence of programming

```
y'+'z"
call'zilch"
prev':'u"
```

will assign the value of the Zilch function of y + z to the variable u.

### 2. Functions as Parameters

When a procedure, such as an integration routine, requires a function as input, it is ordinarily written to include a subroutine to calculate the function, or to call for such a subroutine to calculate the function, or to call for such a subroutine immediately following the procedure call. In order to return to the basic procedure, the initial calling sequence must include a procedure-recall statement. The recall statement consists of the procedure name and the suffix 2'; the arguments are omitted.

As an illustration, suppose we have a procedure (root) which requires as input a tolerance (tol), an initial guess ($y^0$) of the value of the root, and a function (fct). The procedure is to return with the accumulator containing the value of the function fct(y) for the value of y originally in the accumulator. The calling sequence for this procedure might be

```
y0':'y"
call'root'arg'tol'arg'y"
call'fct"
call'root'2".
```

## C. Translation of Procedures

It is essential that a procedure be translated before any call of that procedure. Failure to meet this requirement is not detected by the translator and is the responsibility of the user. A safe rule is to translate all procedures before the main program.

After loading the ACT-IIIA(S) tape and the T-tape, and changing to typewriter input, 6-BIT mode, the first procedure tape is placed in the Flexowriter reader; translated procedures obtained from a library are generally provided on separate lengths of tape, the last operator of which is wait'. When the tape reaches the wait' instruction, the translator stops to allow the tape to be changed. Unless special steps are taken, procedures are not traced, nor are their statement numbers and variables printed in the directories produced at the end of translation.

A final note of caution regarding identification of library procedures, particularly from installations where new procedures are under development. In some instances, a procedure with a given name may exist in several different versions which are not entirely equivalent. Therefore, a program which uses library procedures should include copies of all procedures which it requires or, at least, a reference to the specific procedures that are used.

## XII. ARITHMETIC OPERATIONS WITH INTEGERS

Although arithmetic operations are performed more frequently with real numbers, the ACT-III language provides facilities for performing the corresponding operations with integers. We have already met integers as program constants and as problem parameters, and have learned how to read and write them. It is now appropriate to describe the basic and the special integer operations that are available.

The primary application of integers is for such housekeeping operations as counting, subscripts, and switching. However, their fundamental characteristic, that they are represented exactly in the computer, without error due to roundoff, or conversion to binary fractions, means that they can be used to avoid accumulating this error.

### A. Basic Integer Operations

The basic integer operators are distinguished by the prefix letter i, followed by the symbol used for the floating-point operator. Thus,

i+', i-', ix', and i/'

are the operators for adding, subtracting, multiplying, and dividing integers.

In division, the statement

divd'i/ divr':'quot''

produces a quotient and a remainder of the same sign as the divisor. The remainder is stored as a special variable, remdr'. It can be used later in the program until it is replaced by the remainder from a subsequent division operation.

For the multiplication of small integers (with product less than 134,217,727 in magnitude), a special operator nx' is provided. This is faster than the operator ix' and does not require a special subroutine.

EXERCISE 14

Any common factor of two integers is also a factor of the remainder when the larger of the two is divided by the smaller. With this knowledge, construct a program to print the greatest common denominator of two integers input from the keyboard.

## B. Special Operations

Two special integer operations are also available. The operator iabs', with only a right operand, produces the integer which is the absolute value of the integer right operand. The operator ipwr', with integer right and left operands, produces the integer which is the left operand raised to the right operand power. If the right operand is negative, left operand zero gives an error stop; left operand one gives one; and left operand greater than one gives zero.

EXERCISE 15

Write a program which will read a set of positive and negative integers from the keyboard, and select the one which is largest in magnitude and the one which is smallest in magnitude. Upon exiting from the read phase, the program is to print max for the integer with largest magnitude, min for the integer with smallest magnitude, and $(\mathrm{max})^{\mathrm{min}}$.

## C. Conversion between Integer and Floating-point Numbers

There are several operations which involve both integers and floating-point numbers. Three operators are available to effect conversions. The operator flo', with integer left and right operands, produces a floating-point number which is equal to 0.1 raised to the left operand power multiplied by the integer right operand. Thus,

| | | |
|---|---|---|
| 0'flo'123' | would yield | .123"e'3' |
| 1'flo'123' | would yield | .123"e'2' |
| ['0'i-'1']'flo'123' | would yield | .123"e'4' |

Conversion of floating-point numbers to integers may be accomplished by either of two operators, unflo' or fix'. Both operators require an integer left operand and a floating-point right operand. If the left operand is denoted as n, it converts the right operand, multiplied by $10^n$, to an integer. In the case of unflo', the number to be converted to an integer is rounded after scaling. In the case of fix', the next smaller integer is taken (the next larger in magnitude, if the number is negative). To illustrate, the following results would be obtained with the right operands and operators indicated.

| y | 0' unflo'y' | 0'fix'y' | 2' unflo'y' |
|---|---|---|---|
| 15 734 | 16 | 15 | 1573 |
| 1.826 | 2 | 1 | 183 |
| 2947.301 | 2947 | 2947 | 294730 |
| -1.3275 | -1 | -2 | -133 |
| -1.5275 | -2 | -2 | -153 |

## EXERCISE 16

Give the results of

0'unflo'y', 0'fix'y', 3'unflo'y', 3'fix'y', ['0'i-'2']'unflo'y', ['0'i-'2']'fix'y'

on each of the following numbers, carried in real (floating-point) form:

| | | | |
|---|---|---|---|
| a.) | 0.51635 | e.) | 51.6354 |
| b.) | 0.051635 | f. ) | 51.0000 |
| c.) | 0.00051635 | g.) | $-0.5163542 \times 10^{-3}$ |
| d.) | -51.6354 | h.) | 516,354,200.0 |

## D. Scaling Floating-point Numbers

The final operation which has a fixed-point operand is the operator x10p'. It consists of a floating-point left operand, a fixed-point right operand, and produces a floating-point result. This operator multiplies the left operand by the power of ten given by the right operand. It can be used for scaling if numbers become larger in magnitude than $10^{31}$ or smaller than $10^{-32}$.

## EXERCISE 17

(A) Write a program using floating-point arithmetic to calculate and print the floating-point representation of the numbers from 0 to 100. Print the numbers with format number 1709', five numbers to the line.

(B) Write a second program to compute these numbers by integer arithmetic and floating the integer just before printing.

If possible, translate and run both programs.

## XIII. SUBSCRIPTED VARIABLES

In many problems, we are interested, not in single numbers, but in arrays or ordered groups of numbers. For example, a complex number is usually characterized by two real numbers, its real and its complex parts; a vector in n-dimensional space may be characterized by its n components; a polynomial of degree n in one variable may be specified by its n + 1 coefficients; a system of m homogeneous linear equations in n unknowns may be summarized by the $m \times n$ matrix of coefficients. In all these cases, it would be more convenient to refer to the whole array of numbers by a single name and to use some device to select individual elements. ACT-III provides such a device: subscripted variables.

### A. Dimension Statements

In handling an array, the translator must be informed how much storage to set aside for the elements of the array. This is done by the dimension statement, which has the form: dim', followed by the names and the maximum number of elements in the respective arrays. An example of a dimension statement is

```
dim'poly1'25'poly2'10'mtrix'26".
```

Several arrays may be defined by a single dimension statement, and a program may contain several dimension statements; however, each array must be defined by a dimension statement before it is referred to. If an array or index is given the same name as a previously named variable, the previous definition is erased from the symbol directory. However, all parts of the program which have already been translated will refer to the old variable.

### B. Single Subscripts

Elements of a one-dimensional array, such as a vector or the coefficients of a polynomial, are referred to by the array name, followed by a stopcode. This, in turn, is followed by either a constant integer or a non-negative integer variable, which is the subscript, or index. If the subscript is a variable, a statement is required to the effect that it is to be used as an index. This statement is of the form

```
index'i'j'k".
```

The first element of an array is referred to by the array name and the index 0, the second by the array name with index 1, and so on. Thus the expression a'0' refers to the first element of the array named a. If the dimension of a is 26 or more, a'25' refers to the 26th element. The elements of a are actually stored in reverse order. If the array a is stored in locations 3000 to 3026, the element a'0' is in location 3026, and a'26' in location 3000. If an index is used greater than the dimension of the array, an element is selected from the next-named array or simple variable. On the other hand, if a variable index has not been assigned a value before it is used as a subscript, what Bowlden describes as "mysterious results" may occur.

As an example of the use of subscripts, let us evaluate a polynomial of degree n≤50, the coefficients of which are to be read into the array poly. The coefficient a'i' is the coefficient of $y^i$. The following program will accomplish this:

```
        rdxit's7"
        dim'poly'51"
        index'j"
s11'    0':'j"
        iread'n"
        n'i+'1':'lim"
s1'     read'poly'j"
        j'i+'1':'j"
s4'     if'['j'i-'lim']'neg's1"
s7'     read'y"
        1'i+'n':'j"
        poly'j':'value"
s2'     j'i-'1':'j"
        trn's3"
        value'x'y'+'poly'j':'value"
s5'     use's2"
s3'     cr'1608'print'value"
        use's11'"
```

## EXERCISE 18

Assume that array a, of n elements, contains the elements $a_j$, $0 \le j \le n-1$, of an n-dimensional vector a; also that array b contains the corresponding elements of a vector b. Write a program to compute and print the scalar product (sp) of a and b:

$$sp = \sum_{j=0}^{n-1} a_j b_j \quad .$$

### EXERCISE 19

Construct a program to store the coefficients of the polynomial prod, which is the product of the polynomials poly 1 and poly 2. Assume that poly 1 and poly 2 are of degrees n1 and n2, and that the coefficients are stored in positions corresponding to the exponent.

## C. Incremented Indexes

It is often desirable to refer to sets of elements of an array which are in some fixed relation to each other. In the ACT-III language, if an array name is followed by an index name and an integer program constant, in either order, the sum of the index and the constant is taken as the index. For example, if i = 1,

```
array'i'25' = array'25'i' = array'26'
```

provided, of course, that array has a dimension of 27 or more.

Any two variables, or two variables and a constant, or single variable and an integer constant, which are not separated by an operator are interpreted as a subscripting of the first-named variable. If the second variable has not been declared as an index, an e8 error stop will occur. If the second variable has been defined as an index, the program will be interpreted as written, even if it was not so intended.

### EXERCISE 20

The Bessel function $J_n(y)$ obeys the recurrence formula

$$J_{n-1}(y) = (2n/y)J_n(y) - J_{n+1}(y) \quad .$$

Assume that values of $J_n(y)$ and $J_{n+1}(y)$ are given. Then write a program to compute the values of $J_0(y)$, $J_1(y)$,...$J_{n+1}(y)$ and store them in the array J, with J'0' = $J_0(y)$, etc.

## D. Double Subscripts

Two-dimensional arrays, such as matrices, are defined by the same dimension statement as is used for one-dimensional arrays. However, the elements of such arrays are defined by a double-index statement of the form

```
dbind'ij''.
```

This statement defines a two-element array ij, with elements ij'0' and ij'1. Now, after the (integer) number of columns has been placed in array'0', the value i has been placed in ij'0', and the value j in ij'1', the statement array'ij' will refer to the element in row i and column j of the array.

## EXERCISE 21

Write a program for finding the product of an $(n \times n)$ matrix by an n vector. Include dimension statements, permitting n to be as large as 10, and the necessary index and double-index statements.

## XIV. ITERATIONS

In using subscripted variables, as well as in a number of other applications, we frequently find outselves performing an operation for some value of an integer, which we will call the controlled variable, then changing the integer by a given amount, and repeating the operation until the integer reaches some limit. We did this twice in our polynomial evaluator in Section XIII: the first loop was used to read in the values of the coefficients (statements s1' through s4'); the second loop evaluated the polynomial (statements s2' through s5').

Since this type of calculation occurs so frequently, ACT-III provides a special way to carry it out. In the latest version (T-5) of the compiler, a loop of this kind is created by labeling the first statement of the loop (after initializing the controlled variable and any other variables needed) and placing at the end of the loop the statement

```
for'controlled variable'step'increment'until'limit'rpeat'sX".
```

In this statement, the name of the controlled variable is inserted between <u>for</u>' and <u>step</u>': the amount by which it is to be changed is inserted between step' and <u>until</u>'; the limit which is to be passed to leave the loop is inserted between until'; and <u>rpeat</u>'; and sX' denotes the statement number of the start of the loop.

The controlled variable <u>must be an integer</u>. It may be either a simple integer variable, or a subscripted variable such as one component of a double index. The increment and limit may be integer program constants, simple variables, subscripted variables, or arithmetic expressions. The iteration terminates when the value of (limit-controlled variable) X increment becomes negative. There are no restrictions on sign of the increment or of the limit.

The for' statement

```
for'cvar'step'delta'until'limit'rpeat'sX"
```

produces essentially the same object program as would

```
delta':'temp"

prev'i+'cvar':'cvar"

if'['['prev'i-'limit'i-'temp']'ix'temp']'neg'sX".
```

This object program insures that a zero increment will not cause a perpetual loop.

There is no special instruction in the ACT-III language for iterating with a floating-point controlled variable. On the few occasions when this is desired, two alternatives are possible. The first is to produce the desired floating-point controlled variable by floating an integer with the proper scaling and using a for' statement to increment the integer representation. The second is to write a section of programming equivalent to that produced by the for' statement, but using floating-point arithmetic. For example, suppose that it is required to evaluate some function fct at intervals of 0.01 in the independent variables y. Two ways of accomplishing this would be

```
        0':'y"
s1'     2'flo'y"
        call'fct"
        prev':'temp"
        cr'1602'iprt'y"
        1608'print'temp"
        for'y'step'1'until'100'rpeat's1",
```

or

```
        0':'y"
s1'     y':'temp"
        call'fct"
        prev':'temp"
        cr'1602'dprt'y"
        1608'print'y"
        y'+'.1"e-'1' 'y"
        prev'-'.1005"e'3"
        trn's1"
```

The first form is preferable for several reasons. First, it avoids the inaccuracies due to buildup of roundoff error in repeated addition of 0.01. Secondly, floating-point arithmetic is slower than integer arithmetic. Finally, the last loop would make no provision for the possible vanishing of the increment if it were allowed to vary. A loop with a zero increment is not an uncommon form of programming blunder.

Loops may be used within loops to any desired depth. If a loop is entered by a use' statement to some statement inside the loop, the controlled variable may not have been properly initialized. Caution is indicated

## EXERCISE 22

Write a program for reading the elements of a matrix with m rows and n columns into an array A. The matrix will always obey the condition $m \times n \leq 225$.

## EXERCISE 23

If A is an $(m \times n)$ matrix and B is an $(n \times q)$ matrix, then the product C is an $(m \times q)$ matrix, the (i,j)th element of which is given by

$$C_{i,j} = \sum_{k=1}^{n} a_{ik}\, b_{kj}.$$

Write a program for finding the product of two matrices.

## EXERCISE 24

The binomial coefficients $\binom{n}{m}$ obey the law

$$\binom{n+1}{m} = \binom{n}{m} + \binom{n}{m-1},$$

where

$$\binom{n}{k} = 0 \text{ if } k > n \text{ or if } k < 0$$

and

$$\binom{n}{0} = \binom{n}{n} = 1.$$

Use this information to write a program for computing the binomial coefficients of order nu

## EXERCISE 25

Write a program for computing $n! = 1 \times 2 \times 3 \times \ldots \times n$.

## XV. ADVANCED CONTROL OPERATIONS

In Section VIII (Elementary Control Operations) we learn to use the flow-directing operations use', trn', if', and bkpX'. With these operators in mind (a review might be necessary) we will now proceed to more advanced control operations.

## A. <u>Recalling a Subroutine</u>

In lieu of a procedure, the same section of programming or subroutine can be recalled at several places in the program. To do this, the last instruction in the subroutine must be labeled accordingly. For example, the statement

```
ret'sE'use'sB"
```

may accomplish this purpose, provided sE' is the label of the last statement of the subroutine and is of the form

```
sE'go to's0",
```

and sB' is the label of the first statement of the subroutine. After the statement

```
ret'sE'use'sB"
```

is executed, the statement sE' is changed to use' the statement following use'sB".

## B. <u>Setting Switches</u>

Switches can be set to enable decisions to be made on the flow of a program at a place other than the place where the flow is to be changed. For example, it may be desired to change the course of a loop depending upon some variable which does not change during the loop. It would then be wasteful to test this variable each time the choice had to be made in the loop. The statement

```
set'sE'to'sX",
```

where sE is sE'go to's0", replaces sE by use'sX" and goes on to the statement following the set' statement.

<u>A go to' statement must be preset before it is encountered in executing the program.</u> The operator go to' is not equivalent to the operator <u>use</u>'. Both are translated into an unconditional transfer (u) instruction in the object program. However, in the statement

```
use's0'
```

s0' will be interpreted as the name of a variable and will be assigned a location in the variable storage area. In the statement

```
go to's0'
```

s0' will be interpreted as the name of the statement itself, and the object program will contain an unconditional transfer to the instruction itself. If this instruction is executed before the instruction has been modified by either a set' or ret' statement, the computer will enter a one-word loop. The COMPUTE light will remain on, but none of the registers on the oscilloscope will show any change. This behavior guards against the undetermined actions which might take place if a switch were entered before it had been set.

## EXERCISE 26

A program to compute the Zilch function of the result of the last operation is located between statements s10' and s12'. A program to compute the Nussbaum function is located between statements s20' and s12'. In computing the function u(y), the initial value of y is destroyed. If the initial value of y is positive, it is desired to compute the product Z(w) × N(u); if the initial value of y is negative, the product Z(u) × N(w); or if the initial value of y is zero, the product N(u) × N(w) is to be computed. Use switches to accomplish these results, remembering that the value of y will have been changed before the functions N and Z can be computed.

## C. Indexed Switches

A final device for changing the course of the program depends upon the subscripting. If i has been defined to be an index and if the statement before sT' has the form

```
use'sE'use'sD'use'sC'use'sB'use'sA",
```

the statement

```
use'sT'i"
```

will transfer to sT' if i = 0; to sA' if i = 1; to sB' if i = 2; to sC' if i = 3; to sD' if i = 4; and to sE' if i = 5. If i is outside the limits $0 \leq i \leq 5$, unexpected results may occur.

Only variable subscripts may be used in this way. The statement

```
use'sT'2'
```

will transfer, not to sB', but to statement (T - 2). Similarly, the statement

```
use'sT'i'2'
```

will transfer to the ith-order preceding statement (T - 2).

Subscripted statements cannot be used after go to' or after use' in a ret'sX'use'sY" statement, or after zero' in an if' statement.

## EXERCISE 27

On certain occasions, the computer may be instructed to compute several problems without interruption. Accordingly, the data tape must contain not only the data for the first problem, but that for several problems, not necessarily of the same type. In our exercise, we may wish to treat sets of data on tape in any of four different ways Instructions for the first process start at s10', those for the second at s20', those for the third at s30', and those for the fourth at s40'. Upon completion of each process, the program transfers to s100.

Write a section of programming starting at s100' Give a sample input for the processing of five sets of data. the first by process 1; the second by process 3; the third by process 2; the fourth by process 4, and the fifth by process 2. Also, instruct the computer to stop after the fifth set has been processed.

## D. Calling Procedures

The call' operator used for calling procedures has the same effect as the statement

```
ret'sP'use's(P + 1)",
```

where sP' is the first instruction of the procedure and s(P + 1) is the second Unlike the ret'use' statement, call'proc' can have the address modified by a constant subscript. Thus, the statement

```
call'proc'2"
```

will place the return address in the second instruction before the beginning of proc', and will transfer to the first instruction before it. This technique is useful when it is necessary to leave and reenter a procedure.

# XVI. SPECIAL OUTPUT AND INPUT

The essentials of a definitive program (see Section IX) emphasize the liberal use of explanatory comments in the program itself and, particularly, the assignment of descriptive labels to the input parameters. The same applies to output. Output is more likely to be referred to long after the details of the program which produced it have been forgotten.

Although proper planning of format can do a great deal toward clarifying output, alphabetic text is by far the most effective way of explaining the output layout. ACT-III provides two operators for producing output of all the characters and functions of the Flexowriter keyboard.

## A. Programmed Alphabetic Output

A statement consisting of the operator daprt' followed by a sequence of characters and spaces, each separated by stopcodes, will cause the characters appearing after the operator to be printed. If it is desired to produce the typewriter control functions, the following mnemonic codes must be used:

| | |
|---|---|
| lower case | lc1' |
| upper case | uc2' |
| color shift | color' |
| carriage return | cr4' |
| backspace | bs5' |
| conditional stop | stop' |
| apostrophe | ap' |
| tab | tab6' |

For example, the statement:

```
daprt'cr4'tab6'uc2'E'lc1'x'a'm'p'l'e' 'o'f' 'uc2'D'lc1'i'r'e'c't' 'uc2'

A'lc1'l'p'h'a'b'e't'i'c' 'uc2'P'lc1'r'i'n't'i'n'g"
```

when executed would produce the output

```
Example of Direct Alphabetic Printing.
```

The daprt' operator produces two instructions for each character and thus can consume a large amount of object program space; however, daprt' is fast and simple if the space can be afforded.

## B. Alphabetic Output and Input of Coded Information

A second alphabetic output operator, aprt', requires less program space and prints coded alpha-numeric information which is stored as a variable. Up to five characters or typewriter control functions can be stored as single variable. The variable may be subscripted. For example, the statement

```
aprt'alpha"
```

will cause the alpha-numeric contents of alpha to be printed. If, by error, alpha does not contain coded alpha-numeric information, it will be interpreted as alpha-numeric data, regardless. If some of the characters are not acceptable to the Flexowriter, a print stop may occur. If the variable is a negative number, nothing will be printed.

The alpha-numeric information in the variable may be inserted by the operator aread', followed by the name of the variable in which the alpha-numeric information is to be stored. When this statement is executed, a single word is read from the tape, containing up to four characters of alpha-numeric information in the special code given in Appendix F.

## C. Repeated Alphabetic Output

The operator reprt' can be used to print a consecutive string of identical characters - for example, a line of periods to separate cases of a problem - or to carriage return to the next page. The operator has an integer left operand, giving the number of times the character is to be printed, and a right operand which is the character, or the operation to be repeated. If the left operand is negative, nothing will be printed. For example, the statement

```
5'reprt'cr4"
```

would produce five carriage returns. The reprt' operator with its operands must form a separate statement.

## D. Compatible Output

It is occasionally helpful to punch output on tape in a form which can be accepted later as input to the computer. If it is desirable that the output be legible to the programmer, the operators punch' and ipch' cause the right operands to be printed (and punched if the punch is on) in the form used for floating-point and integer problem parameters, respectively. The numbers being output must obey the restrictions on problem parameters. In particular, if the right operand for ipch' has more than seven digits, an e3 error stop will occur.

## E. Hexadecimal Output and Input

If it is unnecessary for the programmer to understand the intermediate data, as, for example, if the output from one program is to be processed by another, hexadecimal output may be used. The operators are hxpch' and rdhex'. The former causes the right operand to be punched out in hexadecimal format; the latter causes a hexadecimal word to be read and assigned to the right operand. Hexadecimal input and output are faster and more accurate than decimal input and output, since there is no need for binary-decimal conversion, which is slow and inexact.

## F. Read and Float

Occasionally only floating-point operations may be required on a number which is given in integer form on the data tape. This may be accomplished by the two statements:

```
iread'temp"

j'flo'temp':'float".
```

These statements would read an integer from the Flexowriter or reader, store it in temp', then convert it to a floating-point number equal to temp' multiplied by $(0.1)^j$, and store the results in float'. The same result can be obtained by the single instruction

```
j'rdflo'float".
```

## EXERCISE 28

Write a program and include any input data necessary to produce the following output format:

The first line of each page of output is to be labeled with the programmer's name and the date. (This information is to be read in coded form from tape.) The second line is to contain the run number and the page number. (Initial values are to be read from tape; subsequent values are to be assigned consecutively.) The results for each run are to be displayed in sets, each set consisting of three lines of data followed by a blank line. The number of sets is variable. Each run is to start on a new page. The printed page size should measure 84 spaces wide and 66 lines long.

# XVII. WRITING NEW PROCEDURES

Eventually many programmers will want to write their own procedures, either because they need a specialized set of programming which requires more complicated input and results than can be provided by the ret'use' statement, or because they wish to contribute to the library.

## A. Basic Requirements

Each procedure requires an enter' statement, at least one exit' statement, and an end" statement. The translator includes tests to determine that each end" statement has been preceded by an enter' statement, that an exit' statement occurs between each enter' and end" pair, and that a new enter' statement is not made before any previous procedure has been ended. Failure to observe these conditions will cause an e4 error printout.

Ordinarily, a procedure communicates with the remainder of the program through the results of the last operation before entering and leaving it, and by its arguments. Statement numbers may be duplicated between a procedure and the main program. Names of variables also may be duplicated, except for variables named before the operator local', if it appears (see Section D, Global Variables).

Each procedure is prefaced by the operator enter', followed by the name of the procedure, and then by the names to be used for the arguments. When the procedure is called, these names will be replaced by the names in the procedure call. The enter' statement may be preceded by a statement consisting of stop' operators, and of use'0' phrases.

The operator exit' is used to return from the procedure to the main program. It may be used at any place within the procedure.

The operator end" designates the last statement of a procedure. When this statement is read, all statement numbers and variables local to the procedure are erased from the directory, and can no longer be referred to.

## B. References to Arguments

The arguments specified in the procedure call may be arrays. Within the procedure body, all the arguments must be referred to as arrays. Even an argument which is actually a simple variable must be referred to as an array of dimension 1, with the subscript 0 stated explicitly. For example, in the Zilch procedure with entry statement

```
enter'zilch'svar'array"
```

the argument svar' is a simple variable. Within the procedure, svar' must be referred to as svar'0'.

The procedure body must include all necessary definitions of arrays, single indexes, and double indexes for the quantities which are used inside the procedure.

## C. Temporary Exits from a Procedure

If a procedure, such as a quadrature routine or a differential equation routine, requires a function as an argument, the statement before the enter' statement may be used. The call' operator places a return transfer to the statement following the call' statement in the first order of the procedure being called and transfers to the second order. Since SX'i' is the ith location before SX', the statement

```
call'sub'1"
```

occurring somewhere inside the procedure sub' will place the address of the next statement in the location one preceding the beginning of the procedure, and will transfer to the first instruction of the procedure, which returns control to the statement following the subroutine call. To return to the procedure where it was left, the statement call'sub'2" will place the return address in the second location before sub', and will transfer to the first instruction before the procedure. The call'sub'1" statement inside the procedure has already placed the return address in that location. For the return addresses to be useful, they must be inserted into transfer instructions. These may be produced by inserting one use'0' phrase in the statement preceding the enter' statement for every transfer to be made.

The exit' statement is ineffective if the procedure has been left previously by the technique described above. Instead, the final exit is made by the statement use'sub'n", where n is the subscript used for the last call' sub'n" instruction to return to the subroutine from the main program.

If an address was not set in location sub'n+1' by a call'sub'n+1", or a set'sub'n+1", before the statement call'sub'n" or use'sub'n", the program will usually stop in track 62.

The statement before the enter' statement can also be used to store parameters needed by both the procedure and the main program. In this case, a stop' in the statement before the enter' statement will reserve one storage location.

To illustrate, let us suppose that the procedure Zilch requires one intermediate exit to provide a function value, and one temporary storage to be available to both procedure and main program. The beginning of the procedure might be

```
stop'use'0'use'0"
enter'zilch".
```

The intermediate exit would be made from within Zilch by the statement

```
call'zilch'1",
```

the return to Zilch by

```
call'zilch'2",
```

and the final exit by

```
use'zilch'2".
```

The variable would be referred to as zilch'3' by either the main program or the procedure.

## D. Global Variables

The latest version of ACT-III allows an exception to the rules that all variables introduced in a procedure are local to that procedure, and cannot be referred to from outside the procedure, and that a procedure cannot refer to any variables defined outside the procedure. If the operator local' appears in a procedure after the enter' statement, all variables named between the enter' operator and the local' operator are made nonlocal or "global," i.e., they have the same significance inside and outside the procedure. If the local' operator is followed by other names in the same statement, they are interpreted as a continuation of the parameter list from the enter' statement. In this case, only a dim' statement can appear between the enter' statement and the local' operator.

Once a variable name has been identified as global, it remains global for all procedures translated thereafter. (Any other statement translated before a procedure would require a jump to the main program and will result in an error stop when the end" statement following the procedure is translated.) For example, in the following sequence of programming:

```
        enter'zilch"
        dim'a'l'b'l'c'l"        a, b, and c are global variables'
        local'u'v'w"            u, v, and w are the formal parameters'
s1'     a':'r"                  r is local to zilch procedure'
        . . . . . . . .
        exit"
        end"
        enter'beer"
        dim'r'l's'l"            r, s are global variables'
        local'bud'blatz"        bud, blatz are formal parameters'
s2'     bud'0':'a"              this a is a global variable'
s3'     b':'c"
s4'     r':'blatz'0"
        . . . . . . .
        exit"
        end"
        0'.'r"
        1':'a"
        2':'b"
```

```
3':'c"

4':'r"

5':'s"

call'zilch'arg'6'arg'7'arg'8"

call'beer'arg'10'arg'20"

. . . . . . .
```

Global names are useful in allowing simple communication of parameters which will always have the same name between procedure and main program. However, they are not recommended for library procedures or for other procedures which may be used several times in different contexts. It is recommended that any global variables used be listed explicitly in the operating instructions for the procedure, and that wherever possible they be given names distinctive to the procedure. One convention is to use the first three or four letters of the procedure name, followed by a number, letter, or other character. For example nonlocal variables used in the Zilch procedure might be named: zilcl', zilc2', zilca', and so on. The likelihood of unintentional duplication of names of this type is minimal.

## E. Checking Procedures

Checking procedures requires some special consideration. Since ordinarily procedures which are used have already been checked, complete procedures with enter' and end' operators are not normally trace-compiled, regardless of the position of the TRANSFER CONTROL button. It is, of course, helpful to be able to bypass this rule when errors are detected inside a procedure. To trace-compile a procedure the following rules must be obeyed:

(1) The procedure to be checked must have the statement trace" included immediately after the enter' statement. To avoid remaking the tape, it may be typed in from the keyboard. This is done by depressing the MANUAL INPUT lever on the Flexowriter as soon as the second stopcode of the enter' statement has been read.

(2) The TRANSFER CONTROL button must be down. The trace will not include statement numbers within the procedure which are erased when the end" statement is read.

To preserve these statement numbers, the following additional rules must be followed:

(3) The statement before the enter' statement must begin with use'sS', where sS' is the first statement of the main program.

(4) All other procedures necessary must have been translated previously.

(5) There must be no duplication of statement numbers or of local variable names between the procedure being checked and the main program.

(6) The end" statement must be omitted from the procedure being checked.

The conditions which must be observed in checking out a procedure make it advisable to check out each procedure separately from the program in which it is to be used. The effort required to write a small program to provide input, the procedure call, and output to drive the procedure being checked is well spent.

In checking procedures, it is necessary to concentrate attention on one at a time. For complicated programs, this practice is advisable even when it is not enforced by the language. Most experienced programmers find a systematic approach of this sort the best approach to program checkout.

## XVIII. MACHINE OPERATIONS

The operators described in this primer represent combinations of sixteen basic machine operations designed to perform any operation of which the computer is capable. ACT-III provides for the incorporation of machine operations. The operators are:

| | | | |
|---|---|---|---|
| bring' | (b) | hold' | (h) |
| add' | (a) | clear' | (c) |
| subtr' | (s) | stadd' | (y) |
| mult' | (m) | ret' | (r) |
| nmult' | (n) | use' | (u) |
| div' | (d) | stop' | (z) |
| extrt' | (e) | trn' | (t) |

They have right operands, which are the addresses of the machine orders, and leave their results in the accumulator. Their use requires a knowledge of machine language programming, which is beyond the scope of this primer Further information can be obtained by writing POOL, the LGP-30 users' organization.

## XIX. CONCLUSION

Our introduction to the language of the ACT-III compiler is now completed. It is a powerful aid to programming algebraic and scientific problems, and produces object programs which are more efficient than most interpretive routines or unoptimized machine codes. Its scope is, indeed, wider than scientific programs. The inclusion of basic machine operators permits it to be used as a convenient and effective symbolic assembly program, and to express any program which can be programmed for the LGP-30 by any means. Such problems as symbol manipulation, data reduction, and many others fall within the range of the programmer skilled in its use. Further skill in the language must be obtained primarily by practice and experimentation; this is left to the reader.

# APPENDIX A

## Errors at Compile-Time

| Error Printout | Meaning | Remedy |
|---|---|---|
| e1 | Symbol table full (max. 126) | Put some variables into regions |
| e1 | Too many constants (max. 63) | Read in some as data |
| e3 | Incorrect constants | |
| e4 | Improper use of "end," "enter," or "exit" | Correct tape and restart at beginning of statement |
| e5 | Invalid bracket count | |
| e6 | Statement too large | Segment statement and restart at beginning of statement |
| e7 | Statement number too large (max. 191) | Correct tape and restart at beginning of statement |
| e8 | 6-bit button up | Restart at beginning of statement |
| e8 | Invalid subscript | |
| e8 | Invalid operator | |
| e8 | Stopcode missing from previous "dim," "index," "dbind," "enter," or "local" statement | Correct tape and restart at beginning of statement |
| e9 | Invalid or missing operand | |
| s000 0000 | Storage exceeded | Rewrite program |
| sxxx xxxx | Undefined statement | Correct program and recompile |

## APPENDIX B

### Errors at Run-Time

NOTE: Continuing the program after an error display will produce invalid results.

| Operator | Error Type | Meaning |
|---|---|---|
| +, -, x, / | e1 | Floating-point overflow |
| exp, flo, x10p, pwr | e1 | Floating-point overflow |
| / | e2 | Division by zero |
| pwr | e2 | Left operand negative; or left operand zero and right operand negative |
| ln, log | e2 | Operand zero or negative |
| sin, cos | e2 | Operand greater than $10^8$ |
| sqrt | e2 | Operand negative |
| ix, i/, unflo, fix | e3 | Integer overflow |
| ipwr | e3 | Left operand zero and right operand negative |

## APPENDIX C

### ACT-III Operators

| Page | Code | Example | Meaning | Precedence |
|---|---|---|---|---|
| 18 | [ | | Left bracket (maximum of 7) | |
| | ] | | Right bracket (brackets over-rule precedence) | |
| 13 | ; | a';'b' | Substitute value a into b (a unchanged) | 0 |
| 15 | + | a'+'b' | Floating-point addition | 1 |
| 15 | - | a'-'b' | Floating-point subtraction | 1 |
| 17 | 0- | 0-'aa' | Floating-point negation of aa | 3 |
| 15 | x | a'x'b' | Floating-point multiplication | 2 |
| 15 | / | nu'/'den' | Floating-point division | 2 |
| 47 | i+ | n'i+'k' | Integer addition | 1 |
| 47 | i- | n'i-'k' | Integer subtraction | 1 |
| 47 | ix | j'ix'k' | Integer multiplication | 2 |
| 47 | i/ | j'i/'k' | Integer division | 2 |
| 47 | nx | j'nx'k' | Fast integer multiplication for product 134,217,728 | 2 |
| 17 | abs | abs'av' | Absolute value of floating-point av | 3 |
| 48 | iabs | iabs'kk' | Absolute value of integer kk | 3 |
| 48 | flo | n'flo'b' | Generate floating-point equivalent of integer b with last n digits fractional (b unchanged) | 3 |
| 48 | unflo | j'unflo'b' | Generate rounded integer equivalent of floating-point b with decimal moved j places right | 3 |
| 48 | fix | j'fix'b' | Unfloat but drop fractional digits | 3 |
| 48 | ipwr | a'ipwr'n' | Integer a to integer n'th power | 3 |
| 49 | x10p | a'x10p'n' | Move decimal point of floating-point a, n places right | 3 |

## INPUT-OUTPUT

| Page | Code | Example | Meaning | Precedence |
|---|---|---|---|---|
| 23 | read | read'a" | Read floating-point number and store in a | 0 |
| 61 | punch | punch'a" | Punch floating-point a with conditional stops for input | 0 |
| 24 | print | n'print'a" | (n = 100c + s) print a as a floating-point number in c columns, rounded to s significant digits | 0 |
| 25 | dprt | i'dprt'a" | (i = 100c + s) print floating-point a as decimal number in c columns with s fractional digits | 0 |
| 23 | iread | iread'a" | Read an integer number and store in a | 0 |
| 61 | rdflo | n'rdflo'a" | Read integer, convert it to a floating-point value with last n digits fractional, store in a | 0 |
| 61 | ipch | ipch'n" | Punch integer n with conditional stop for input | 0 |
| 25 | iprt | n'iprt'i" | (n = 100c + f) print integer i in minimum of c columns with f fractional digits (f not exceeding 8) | 0 |
| 61 | aread | aread'b" | Read one word in alphabetic code into b | 0 |
| 60 | aprt | aprt'b" | Print b as alphabetic information | 0 |
| 60 | daprt | daprt'n'e'g" | Print specific characters; example, neg | * |
| 61 | reprt | n'reprt'cr4" | Print individual character or control n times | * |
| 61 | hxpch | hxpch'a" | Punch a as a hexadecimal word with conditional stop for input by rdhex | 0 |
| 61 | rdhex | rdhex'a" | Read a hexadecimal word and store in a | 0 |
| 31 | rdxit | rdxit's13" | Data input terminates when a blank word is read; control is transferred to s13' | 0 |
| 26 | cr | cr' | Execute typewriter carriage return | 0 |
| 26 | tab | tab' | Execute typewriter tab | 0 |

*Precedence does not apply

## CONTROL

| Page | Code | Example | Meaning | Precedence |
|---|---|---|---|---|
| 29 | trn | trn's7" | Transfer control to s7' if accumulator neg. | 0 |
| 29 | use | use's8" | Transfer control to s8', regardless | 0 |
| 32 | stop | stop" | STOP! Continue if "START" pressed on console | 0 |
| 57 | ret<br>go to | ret's2'<br>use's1" | Transfer to s1' after storing return address at s2', written s2'go to's0" | 0 |
| 57 | set<br>to | set's2'<br>to's72" | S2', of the form s2'go to's0", is made to read s2'use's72" | 0 |
| 33 | bkp4<br>bkp8<br>bkp16<br>bkp32 | bkp4'<br>use's2" | For machines with overflow logic mod. only; transfer control to s2' if the bkpt. switch is down (on) otherwise to the next sequential statement | 0 |
| 34 | oflow | oflow'<br>use's2" | Transfer to s2' if overflow occurred during preceding i+ or i- (overflow logic mod. only) | 0 |
| 54 | for<br>step<br>until<br>rpeat | for'm'<br>step'd'<br>until'j'<br>rpeat's3" | Increase integer m by integer d, transfer to s3' if the new value of m is not greater than j, otherwise to next statement | 0 |
| 29 | if<br>neg<br>zero<br>pos | if'a'neg's1"<br>if'a'neg's1'zero's2"<br>if'a'neg's1'pos's3"<br>if'a'neg's1'zero's2'pos's3" | If floating-point or integer a is neg, transfer control to s1' if zero to s2', if pos to s3', or next statement if tests fail | 0 |
| 14 | prev | prev'-'cv' | Last result to be the operand (must be first operation executed in the statement) | 3 |
| 50 | index | index'k'n" | Set up k and n for use as subscripts (maximum of 30) | * |
| 52 | dbind | dbind'i'j" | Set up i, j for use as double subscripts | * |
| 50 | dim | dim'coef'10'<br>bn'44" | Reserve 10 sequential locations for coef region, 44 for bn region | * |

*Precedence does not apply

| Page | Code | Example | Meaning | Precedence |
|---|---|---|---|---|
| **FUNCTIONS** | | | | |
| 17 | sqrt | sqrt'a' | Square root of floating-point a | 3 |
| 17 | ln | ln'a' | Natural logarithm of floating-point a | 3 |
| 17 | log | log'a' | Common logarithm of floating-point a | 3 |
| 17 | exp | exp'a' | E raised to the floating-point a'th power | 3 |
| 17 | pwr | a'pwr'b' | Floating-point a raised to floating-point b'th power | 3 |
| 17 | sin | sin'a' | Sine of (floating-point a in radians) | 3 |
| 17 | cos | cos'a' | Cosine of (floating-point a in radians) | 3 |
| 17 | artan | artan'b' | Floating-point angle in radians whose tangent = b | 3 |
| 17 | randm | randm' | Generate pseudo-random floating-point value between 0 and 1 | 3 |
| **SUBROUTINE OPERATIONS** | | | | |
| 62 | enter | enter'calc' b12" | Denotes start of source language subroutine named calc b12 is a dummy symbol which refers to a sequential block of data specified in the main program-calling sequence | * |
| 59 | call arg | call'calc' arg'a" | Main program-calling sequence which transfers to the subroutine named calc and makes dummy symbol b12 mean and refer to actual region a | 0 |
| 65 | local | local" | Denotes that variable names preceding local in the subroutine are global | * |
| 62 | exit | exit' | Return control from subroutine to main program | * |
| 62 | end | end" | Denotes end of source language subroutine | * |
| 66 | trace | trace" | Subroutine will be trace compiled if TRANSFER CONTROL button is down | * |
| 46 | wait | wait' | Suspends compilation | * |
| 47 | remdr | remdr' | (Special symbol) remainder of previous i/ operation | |

*Precedence does not apply

## APPENDIX D

### Summary of Operations

Listed below are the most generally used "button pushing" operations for ACT-III. The steps marked with an asterisk (*) pertain to console buttons and those unmarked either to the Flexowriter or to the reader-punch (for use with the ACT-III composite system tape).

READ COMPILER
Compiler tape in photoreader
Source in flex
I Sel rdr. O flex
*O C.N.S.

COMPILE SOURCE PROGRAM
I, O Sel flex
* 6-Bit, T.C. down
Start

PUNCH HEX TAPE
(H.S. punch)
I Sel flex
Flex manual down
*O.C.N S. doat2900'
Start O Sel punch
*Bkp 32 down Start
(Flex)
I, O Sel flex
Flex manual down
*O.C.N S.
doat2900' Start
Flex punch on - identify
Start

READ RUNNING TAPE P5B
P5B in photoreader
I Sel rdr. O flex
Data in flex O.C.N.S.*

PROGRAM IN MEMORY
I, O Sel flex
Flex manual down
*O.C.N.S. doat0300'
Start Flex manual up
Start

PROGRAM NOT IN MEMORY
I Sel reader, O flex
Read P5B and Hex tapes
I Sel flex
Flex manual down O.N.S.*
Flex manual up

RESET COMPILER
T-Tape in flex
I, O Sel flex
*O.C.N.S.

LEGEND

| | |
|---|---|
| I, O | Input, Output |
| Sel | Selector switch |
| rdr | photoreader |
| flex | typewriter or Flexowriter |
| T.C | Transfer control button |
| Bkp | Breakpoint button |
| doat2900'/0300' | typed from keyboard |
| O.C N.S | One operation, clear counter, normal, start compute buttons |
| O.N.S. | One operation, normal, start compute buttons |

## APPENDIX E

### ACT-III Operators and Decimal Memory Print

The first two codes (underlined) below are a jump to the statement stop routine, and a jump to the first executed instruction of the program. They are found on every ACT-III object program.

```
                                  0300  u6048 u0302

s1'  a';'b"                       0302  ,00111qwj  h3062
s2'  a'+'b';'c"                   0304  ,00211qwj  h5336  b3062  r4813  u4600  h3061
s3'  a'+'.3141'59'e'1';'c"        0310  ,00311qwj  h5336  b6200  r4813  u4600  h3061
s4'  a'-'b';'c"                   0316  ,00411qwj  h5336  b3062  r4813  u4835  h3061
s5'  a'x'b';'c"                   0322  ,00511qwj  h5336  b3062  r4813  u4807  h3061
s6'  a'/'b';'c"                   0328  ,00611qwj  h5336  b3062  r4813  u4707  h3061
s7'  a'print'b"                   0334  ,00711qwj  h5336  b3062  r5506  u5130
s8'  a'dprt'b"                    0339  ,00811qwj  h5336  b3062  r5506  u5201
s9'  read'a"                      0344  ,00933034  u4900  h3063
s10' a'i+'b';'c"                  0347  ,00f11qwj  a3062  h3061
s11' a'i+'1';'c"                  0350  ,00g11qwj  a6201  h3061
s12' a'i-'b';'c"                  0353  ,00j11qwj  s3062  h3061
s13' a'ix'b';'c"                  0356  ,00k11qwj  h5336  b3062  r4813  u4200  h3061
s14' a'i/'b';'c"                  0362  ,00q11qwj  h5336  b3062  r4813  u4300  h3061
s15' a'nx'b';'c"                  0404  ,00w11qwj  n3062 m5952  h3061
s16' a'ipwr'b';'c"                0408  ,01011qwj  h5336  b3062  r5506  u4500  h3061
s17' a'x10p'b';'c"                0414  ,01111qwj  h5336  b3062  r4813  u4510  h3061
s18' use's1"                      0420  ,012f0308
s19' for'a'step'1'until'b'        0421  ,01313q04  h5739  a3063  h3063  s3062  s5739
     rpeat's1"                                  m5739  t0302
s20' wait'                        0429  ,02933f0j
```

The next six statements are traced'

```
s21' iread'a"                     0429  ,02933f0j  u5821  r5506  u5500  h3063
s22' i'iprt'a"                    0434  ,01633f0j  u5821  b3060  h5336  b3063  r5506
                                                 u5552
s23' abs'a';'b"                   0441  ,01733f0j  u5821  b3063  r5506  t5161  h3062
```

```
s24' iabs'a';'b"            0447  ,01833f0j  u5821  b3063  r5506  t5740  h3062

s25' punch'a"               0453  ,01933f0j  u5821  b3063  r5506  u5137

s26' wait'                  0458  ,03511qwj

End of trace '

s27' ipch'a"                0458  ,03511qwj  r5506  u6000

s28' aread'a"               0461  ,01j33718  u5861  h3063

s29' aprt'a"                0500  ,01k11qwj  r5506  u5759

s30' if'a"                  0503  ,01q11qwj

s31' if'a'neg'sl"           0504  ,01w11qwj  t0302

s32' if'a'neg'sl'zero'sl"   0506  ,02011qwj  t0302  t0511  s6109  t0302

s33' if'a'neg'sl'zero'sl'   0511  ,02111qwj  t0302  t0516  s6109  t0302 m5662
     pos'sl"                                 t0302

s34' if'a'zero'sl'pos'sl"   0518  ,02211qwj  t0522  s6109  t0302 m5662  t0302

s35' cr"                    0524  ,02381000  z0000

s36' tab"                   0526  ,02481800  z0000

s37' a'flo'b';'c"           0528  ,02511qwj  h5336  b3062  r4813  u4528  h3061

s38' prev'+'a';'b"          0534  ,026j3590  b3063  r4813  u4600  h3062

s39' a'rdflo'b';'c"         0539  ,02711qwj  h5336  r5506  u5500  r4813  u4528
                                             h3062  h3061

s40' a'unflo'b';'c"         0547  ,02811qwj  h5336  b3062  r5506  u4414  h3061

s41' a'fix'b';'c"           0553  ,02911qwj  h5336  b3062  r5506  u4207  h3061

s42' dim'a'10"              0559  ,055f0600

s43' index'jk"              0559  ,055f0600  u0561  r5635  u5600 -r5602

s44' dbind'ij"              0600  ,02jf0614  r5619  u5607  z4600 -r5609

s45' daprt'd'a'p'r't"       0605  ,02k81500  z0001  p5700  z0001  p3300  z0001
                                             p1300  z0001  p4500  z0001

s46' a'reprt'cr4"           0615  ,02q11qqj m5662  t0620  u0624  z3200  p1600
                                             a5809  t0619  h6309  z3200

s47' stop"                  0625  ,02w00000

s48' rdxit'sl"              0626  ,03033620  u0629  u0302

s49' ret'sl'use's2"         0629  ,03130308  u0304
```

```
s50' go to's0"            0631  ,032f067j
s51' trn's1"              0632  ,033g0308
s52' sqrt'a';'b"          0633  ,0341lqqj  r4813  u3100  h3062
s53' ln'a';'b"            0637  ,0351lqqj  r4813  u3700  h3062
s54' log'a';'b"           0641  ,0361lqqj  r5506  u3807  h3062
s55' exp'a';'b"           0645  ,0371lqqj  r4813  u3900  h3062
s56' a'pwr'b';'c"         0649  ,0381lqqj  h5336  b3062  r5506  u3800  h3061
s57' sin'a';'b"           0655  ,0391lqqj  r4813  u3400  h3062
s58' cos'a';'b"           0659  ,03f1lqqj  r4813  u3500  h3062
s59' artan'a';'b"         0663  ,03g1lqqj  r4813  u3200  h3062
s60' randm';'a"           0703  ,03j12q94  h4801  b5254  n4947 m5629  e4834
                                           h5254  r4813  u5003  h3059
s61' set's1'to's2"        0713  ,03k30308 u0716  u0304
s62' bkp4"                0716  ,83q00400 u0719
s63' bkp8"                0718  ,83w00800 u0721
s64' rdhex'a"             0720  ,040k3590 p0000  i0000  h3059
s65' call'sub'arg'a"      0724  ,04131qj4  u3050  z3059
s66' hxpch'a'"            0727  ,0421lqqj  r5506  u6037  z0000
```

Note: The 0-' operator consists of a r5506 u5161.

## APPENDIX F

### Codes for "aread"

| Symbol | Code | Symbol | Code |
|---|---|---|---|
| )0 | 04 | Aa | 72 |
| Ll | 0j | Bb | 0f |
| *2 | 14 | Cc | 6f |
| "3 | 1j | Dd | 2f |
| Δ4 | 24 | Ee | 4f |
| %5 | 2j | Ff | 54 |
| $6 | 34 | Gg | 5j |
| π7 | 3j | Hh | 62 |
| Σ8 | 44 | Ii | 22 |
| (9 | 4j | Jj | 64 |
| Space | 06 | Kk | 6j |
| _ ‾ | 0a | Ll | 0j |
| =+ | 16 | Mm | 3f |
| : ; | 1a | Nn | 32 |
| ?/ | 26 | Oo | 46 |
| ] . | 2a | Pp | 42 |
| [ , | 36 | Qq | 74 |
| Tab | 30 | Rr | 1f |
| Lower Case | 08 | Ss | 7f |
| Upper Case | 10 | Tt | 5f |
| Color Shift | 18 | Uu | 52 |
| Carr. Return | 20 | Vv | 3a |
| Back Space | 28 | Ww | 7j |
| ' | 40 | Xx | 4a |
|  |  | Yy | 12 |
|  |  | Zz | 02 |

## APPENDIX G

### Solutions to Exercises

Programming problems seldom have a unique solution. The solutions may vary in directness, in accuracy, and in efficiency, as measured by speed and storage requirements. If your solutions differ from the ones given here, compare them in these respects; it may well be that your solutions are better. The real test of a program is whether it computes without error indication and produces the desired results.

Carriage returns and tabs are used between statements throughout the program portion of these solutions.

### EXERCISE 1

| | Integer Value | Program Constant | Data or Problem Parameter |
|---|---|---|---|
| a.) | 1 | 1' or +1" | +1' |
| b.) | 321456 | +3214'56' | +321456' |
| c.) | -52 | Negative | -52' |
| d.) | 536,870,911 | +5368'70911' | Too large |
| e.) | -536,870,911 | Negative | Too large |
| f.) | 0 | 0' or +0" | +0' |
| g.) | 742,125,000 | Too large | Too large |
| h.) | 3.1416 | Not an integer | Not an integer |

### EXERCISE 2

| | Program Constant | Data or Problem Parameter | Integer Value |
|---|---|---|---|
| a.) | No sign allowed | +0' | 0 |
| b.) | Negative | -1' | -1 |
| c.) | +1234" | +1234' | 1,234 |
| d.) | +1'23456' | +123456 | 123,456 |
| e.) | 1' | +1' | 1 |
| f.) | Sign needed | +102' | 102 |
| g.) | Too large | Too large | 700,000,000 |
| h.) | Negative | -7000000' | -7,000,000 |

## EXERCISE 3

| | Number | Program Constant | Data or Problem Parameter |
|---|---|---|---|
| a.) | 0 | 0' | 0' |
| b.) | 15.0 | .15"e'2' | +15'+2' |
| c.) | $6.02 \times 10^{23}$ | .602''e'24' | +602'+24' |
| d.) | $-3.00 \times 10^{10}$ | Negative | -300'+11' |
| e.) | 3.14159265 | .3141'59265'e'1' | +3141593'+1' |
| f.) | $5.3 \times 10^{31}$ | Too large | Too large |
| g.) | $-.195 \times 10^{-32}$ | Negative | -195'-32' |
| h.) | $.253 \times 10^{-32}$ | .253"e-'32' | +253'-32 |

## EXERCISE 4

| | Program Constant | Number | Data or Problem Parameter |
|---|---|---|---|
| a.) | .512'34678'e'5' | 51,234.678 | +5123468'+5' |
| b.) | .5"e-'32' | $.5 \times 10^{-32}$ | +5'-32' |
| c.) | Exponent too large | $.7 \times 10^{32}$ | Too large |
| d.) | Negative | -.4 | -4'+0' |
| e.) | Fraction has sign | $.512 \times 10^{5}$ | +512'+5' |
| f.) | Exponent sign should follow e | $.512 \times 10^{-5}$ | +512'-5' |
| g.) | Only three words | $.512 \times 10^{-5}$ | +512'-5' |
| h.) | Six characters in first word | .512342678 | +5123427'+0' |

## EXERCISE 5

| Data or Problem Parameter | Number | Program Constant |
|---|---|---|
| a.) +0'+0' | 0 | 0 |
| b.) More than seven digits | $-.123456789 \times 10^{-5}$ | Negative |
| c.) -12'-2' | $-.12 \times 10^{-2}$ | Negative |
| d.) +123456'+7' | $.123456 \times 10^{7}$ | .1234'56'e'7' |
| e.) +1230000'+7' | $.1230000 \times 10^{7}$ | .123"e'7' |
| f.) Leading zeros | $.123 \times 10^{3}$ | .123"e'3' |
| g.) More than seven digits | $.123456789 \times 10^{-5}$ | .1234'56789'e-'5' |
| h.) Decimal point | $.1234567 \times 10^{-1}$ | .1234'567'e-'1' |

## EXERCISE 6

Examples a.), d.), f.), g.), and h.) represent acceptable names for simple variables.

The other examples are unacceptable: b.) has more than five characters; c.) is a multiplication operator; e.) is a sine operator; i.) is a statement label; and j.) is a constant.

## EXERCISE 7

The values of the variables after each of the statements are:

| | a | b | c | templ |
|---|---|---|---|---|
| 0':'a" | 0 | - | - | - |
| 1':'b" | 0 | 1 | - | - |
| 2':'c" | 0 | 1 | 2 | - |
| a':'templ" | 0 | 1 | 2 | 0 |
| b':'a" | 1 | 1 | 2 | 0 |
| c':'b" | 1 | 2 | 2 | 0 |
| templ':'c" | 1 | 2 | 0 | 0 |

## EXERCISE 8

a.) .1 + (.2 x .8) = .26

b.) (.1 / .2) + (.8 x .4) = .82

c.) (.1 x .2) (1/.8) (.4) = .01

d.) (.1 / .2) / .8 = .625

e.) .1 - .2 x .8 / .4 = -.3

## EXERCISE 9

a.)
```
['['['0-'z'x'.25"e'0'+'.3333'3333'e'0']'x'z'-'.5000"e'0']'x'z'
+'.9999'99999'e'0']'x'z':'res"
```

b.)
```
z'/'['.9999'99999'e'0'+'z'/'['.2"e'1'+'z'/'['.3"e"+.2000"e'0'
x'z']']']':'res"
```

c.)
```
z'x'['.1111'11111'e'0'+'.1888'8889'e'1'/'['z'+'.2431'3725'e'1'-'
.4805'8439'e'0'/'['z'+'.1568'6275'e'1']':'res"
```

d.)
```
z'x'.1111'11111'e'0'+'.1888'8889'e'1'-'.4592'5926'e'1'/'['z'+'
.2629'0323'e'1'-'.2709'8508'e'0'/'['z'+'.1370'9677'e'1']']':'res"
```

The comparison of the expressions is shown in the following table:

| . | +,- | x,/ | Constants | Remarks |
|---|---|---|---|---|
| a.) | 3 | 4 | 4 | Poor approximation |
| b.) | 3 | 4 | 4 | |
| c.) | 4 | 3 | 5 | Best |
| d.) | 5 | 3 | 6 | Difference of two relatively large numbers. |

## EXERCISE 10

```
sqrt'['ex'x'ex'+'y'x'y']':'rho"
artan'['y'/'ex']':'phi"
rho'x'cos'phi':'ex"

rho'x'sin'phi':'y"
ex1'+'ex2':'sumrl"
y1'+'y2':'sumim"
ex1'-'ex2':'difrl"
y1'-'y2':'difim"
ex1'x'ex2'-'y1'x'y2':'prdrl"
```

```
exl'x'y2'+'ex2'x'yl':'prdim"
exl'x'ex2'+'yl'x'y2':'arg"
['exl'x'ex2'+'yl'x'y2']'/'arg':'qotrl"
['ex2'x'yl'-'exl'x'y2']'/'arg':'qotim"
```

## EXERCISE 11

| | n | b | n'iprt'n' | n'print'b' | n'dprt'b' |
|---|---|---|---|---|---|
| a.) | +0000000'+1234567'-5' | | 0 | . e-05 | . |
| b.) | +0001605'+1234567'-5' | | 0.01605 | .12346 e-05 | .00000 |
| c.) | +0000802'+1234567'-5' | | 8.02 | .1 e-05 | .00 |
| d.) | +0000802'+1234567'+0' | | 8.02 | .1 e 00 | .12 |
| e.) | +0000200'+1234567'+0' | | 200 | . e 00 | . |
| f.) | +0000202'-1234567'+5' | | 2.02- | . e 05 | -12346. |
| g.) | +0001608'+1234567'+5' | | 0.00001608 | .12345672 e 05 | 12345.67211940 |

In the interest of clarity, the output above has been arranged in columns. Below is the output as the program has been written.

```
Ex. 11

a. +0000000'+1234567'-5' 0 .  e-05 .
b. +0001605'+1234567'-5'          0.01605      .12346 e-05          .00000
c. +0000802'+1234567'-5'     8.02 .1 e-05      .00
d. +0000802'+1234567'+0'     8.02 .1 e 00      .12
e. +0000200'+1234567'+0' 200 .  e 00 .
f. +0000202'-1234567'+5' 2.02-.  e 05-12346.
g. +0001608'+1234567'+5'        0.00001608  .12345672 e 05 12345.67211940
```

## EXERCISE 12

```
      .45"e'1':'temp"            0':'deg"
s1'   deg'x'.1745'32925'e-'1':'rad"          convert degrees to radians'
      cr'1000'dprt'deg"          cos'rad':'temp"
      if'temp'zero's95"                      skip division by zero'
      1605'dprt'['.9999'99999'e'0'/'temp']"  print secant'
s5'   sin'rad':'temp"
      if'temp'zero's96"                      skip division by zero'
      1605'dprt'['.9999'99999'e'0'/'temp']"  print cosecant'
s6'   deg'+'.9999'99999'e'0':'deg"           prev'-'test':'temp"
      trn's1"          cr'deg'+'.45"e'1':'test"        use's1"
s95'  daprt' ' ' ' ' 'u'n'b'o'u'n'd'e'd' ' ' '         use's5"
s96'  daprt' ' ' ' ' 'u'n'b'o'u'n'd'e'd' ' ' '         use's6"'
```

## EXERCISE 13

The routine at s100 has been assumed to be f(y) = y**p. Input for p has been added.'

```
s1'   read'p"              read'a"          prev':'y"
      read'b"              read'n"          ['b'-'a']'/'n':'incr"
s2'   use's100"
s3'   if'y'-'a'zero's10"                    jump for first point.'
s4'   y'+'incr':'y"
      if'b'-'y'neg's15"    test after incrementing because a test for
                           equality might fail due to round off'
      sum'+'f':'sum"       use's2"
s10'  0-'f'x'.5''e'0':'sum"                 use's4"
s15'  1608'print'['['sum'+'f'x'.5"e'0']'x'incr']"
      use's1"                               return for new integral'
s100' y'pwr'p':'f"         use's3"'
```

```
           Test Data                                         Output

p  +1000010'+1'    a    +0000000'+0'
b  +1000000'+1'    n    +1000000'+2'                  .49999720 e 00

p  +1000000'+2'    a    +0000000'+0'
b  +1000000'+2'    n    +1000000'+3'                  .90991443 e 00
```

## EXERCISE 14

Greatest Common Divisor'

```
s1'   iread'sml i"          iread'lrg i"
s2'   prev'i/'sml i"        if'remdr'zero's3"    sml i';'tempi"
      remdr';'sml i"        bring'tempi"         use's2"
s3'   cr'1000'iprt'sml i"   use's1"
s4'   stop'use's1'"
```

## EXERCISE 15

New numbers are tested and assigned until a blankword is read - then the print-out is made.'

```
      rdxit's10"
s11'  iread'max"                      iabs'max':'abmax':'abmin"
s1'   iread'new"                      iabs'new':'abnew"
      if'abmax'i-'abnew'neg's5"       if'abnew'i-'abmin'neg's6"     use's1"
s5'   new':'max"                      abnew':'abmax"                use's1"
s6'   new':'min"                      abnew':'abmin"                use's1"
s10'  cr'1000'iprt'max"               1000'iprt'min"
      1000'iprt'['max'ipwr'min']"     use's11'"
```

## EXERCISE 16

```
s1'   read'y"                         cr'1200'iprt'['0'unflo'y']"
      1200'iprt'['0'fix'y']"          1200'iprt'['3'unflo'y']"
      1200'iprt'['3'fix'y']"          1200'iprt'['['0'i-'2']'unflo'y']"
      1200'iprt'['['0'i-'2']'fix'y']" use's1'"
```

| | | | | | | ['0'i-'2'] | |
|---|---|---|---|---|---|---|---|
| | y | 0'unflo | 0'fix | 3'unflo | 3'fix | unflo | fix |
| a.) | .51635 | 1 | 0 | 516 | 516 | 0 | 0 |
| b.) | .051635 | 0 | 0 | 52 | 51 | 0 | 0 |
| c.) | .00051635 | 0 | 0 | 1 | 0 | 0 | 0 |
| d.) | -51.6354 | -52 | -51 | -51635 | -51635 | -1 | 0 |
| e.) | 51.6354 | 52 | 51 | 51635 | 51635 | 1 | 0 |
| f.) | 51.0000 | 51 | 50 | 51000 | 50999 | 1 | 0 |
| g.) | -.0005163542 | 0 | 0 | -1 | 0 | 0 | 0 |
| h.) | 516354200 | Error 3 unflo stop - Number too large | | | | | |

## EXERCISE 17(A)

```
       Note error between 0 and 100'

       0';'n"

s1'    cr'5';'crtn"

s2'    1709'print'n"    n'+'.9999'99999'e'0';'n"    .105'00000'e'3'-'n';'ntest"

       trn's3"          crtn'i-'1';'crtn"           if'crtn'zero's1'pos's2"

s3'    stop"'
```

Output

```
.000000004 e 00  .999999945 e 00  .199999992 e 01  .299999956 e 01  .399999921 e 01
.499999885 e 01  .599999849 e 01  .699999813 e 01  .799999778 e 01  .899999742 e 01
.999999706 e 01  .109999959 e 02  .119999949 e 02  .129999940 e 02  .139999930 e 02
.149999921 e 02  .159999911 e 02  .169999902 e 02  .179999892 e 02  .189999883 e 02
.199999873 e 02  .209999863 e 02  .219999854 e 02  .229999844 e 02  .239999835 e 02
.249999825 e 02  .259999816 e 02  .269999806 e 02  .279999797 e 02  .289999787 e 02
.299999778 e 02  .309999768 e 02  .319999759 e 02  .329999749 e 02  .339999740 e 02
.349999730 e 02  .359999720 e 02  .369999711 e 02  .379999701 e 02  .389999692 e 02
.399999682 e 02  .409999673 e 02  .419999663 e 02  .429999654 e 02  .439999644 e 02
.449999635 e 02  .459999625 e 02  .469999616 e 02  .479999606 e 02  .489999596 e 02
.499999587 e 02  .509999577 e 02  .519999568 e 02  .529999558 e 02  .539999549 e 02
.549999539 e 02  .559999530 e 02  .569999520 e 02  .579999511 e 02  .589999501 e 02
.599999492 e 02  .609999482 e 02  .619999472 e 02  .629999463 e 02  .639999453 e 02
.649999444 e 02  .659999434 e 02  .669999425 e 02  .679999415 e 02  .689999406 e 02
.699999396 e 02  .709999387 e 02  .719999377 e 02  .729999368 e 02  .739999358 e 02
.749999349 e 02  .759999339 e 02  .769999329 e 02  .779999320 e 02  .789999310 e 02
.799999301 e 02  .809999291 e 02  .819999282 e 02  .829999272 e 02  .839999263 e 02
.849999253 e 02  .859999244 e 02  .869999234 e 02  .879999225 e 02  .889999215 e 02
.899999205 e 02  .909999196 e 02  .919999186 e 02  .929999177 e 02  .939999167 e 02
.949999158 e 02  .959999148 e 02  .969999139 e 02  .979999129 e 02  .989999120 e 02
.999999110 e 02
```

## EXERCISE 17(B)

```
      Note error between 0 and 100.'

      0';'n"

s4'   cr'5';'crtn"

s5'   0'flo'n';'nflo"           1709'print'nflo"     n'i+'1';'n"

      100'i-'n';'ntest"         trn's6"              crtn'i-'1';'crtn"

      if'crtn'zero's4'pos's5"

s6'   stop"'
```

Output

```
.000000004 e 00   .999999945 e 00   .199999992 e 01   .300000016 e 01   .399999980 e 01
.500000004 e 01   .600000028 e 01   .699999992 e 01   .800000016 e 01   .899999980 e 01
.999999945 e 01   .110000018 e 02   .120000009 e 02   .129999999 e 02   .139999990 e 02
.149999980 e 02   .160000030 e 02   .170000021 e 02   .180000011 e 02   .190000002 e 02
.199999992 e 02   .209999983 e 02   .219999973 e 02   .230000023 e 02   .240000014 e 02
.250000004 e 02   .259999995 e 02   .269999985 e 02   .279999976 e 02   .290000026 e 02
.300000016 e 02   .310000007 e 02   .319999997 e 02   .329999987 e 02   .339999978 e 02
.350000028 e 02   .360000018 e 02   .370000009 e 02   .379999999 e 02   .389999990 e 02
.399999980 e 02   .410000030 e 02   .420000021 e 02   .430000011 e 02   .440000002 e 02
.449999992 e 02   .459999983 e 02   .469999973 e 02   .480000023 e 02   .490000014 e 02
.500000004 e 02   .509999995 e 02   .519999985 e 02   .529999976 e 02   .540000026 e 02
.550000016 e 02   .560000007 e 02   .569999997 e 02   .579999987 e 02   .589999978 e 02
.600000028 e 02   .610000018 e 02   .620000009 e 02   .629999999 e 02   .639999990 e 02
.649999980 e 02   .659999971 e 02   .670000021 e 02   .680000011 e 02   .690000002 e 02
.699999992 e 02   .709999983 e 02   .719999973 e 02   .730000023 e 02   .740000014 e 02
.750000004 e 02   .759999995 e 02   .769999985 e 02   .779999976 e 02   .790000026 e 02
.800000016 e 02   .810000007 e 02   .819999997 e 02   .829999987 e 02   .839999978 e 02
.850000028 e 02   .860000018 e 02   .870000009 e 02   .879999999 e 02   .889999990 e 02
.899999980 e 02   .910000030 e 02   .920000021 e 02   .930000011 e 02   .940000002 e 02
.949999992 e 02   .959999983 e 02   .970000033 e 02   .980000023 e 02   .990000014 e 02
.999999945 e 02
```

## EXERCISE 18

```
      Scalar Product'

      dim'a'51'b'51"        index'j"

s1'   iread'n"              0':'j"

s2'   read'a'j"             j'i+'1':'j"          if'prev'i-'n'neg's2"

      0':'j"

s3'   read'b'j"             j'i+'1':'j"          if'prev'i-'n'neg's3"

      0':'sp':'j"

s4'   sp'+'a'j'x'b'j':'sp"  j'i+'1':'j"

      if'prev'i-'n'neg's4"  1608'print'sp"       use's1"'
```

## EXERCISE 19

The coefficient prod, i, is the sum of products pl, j x p2, i - j, where j runs between the greater of i - n2 and 0, and the lesser of i and nl. (For statement used.)'

```
     dim'polyl'64'poly2'64'prod'128"      index'i'j'i-j"
s3'  rdxit's2"                            To read in the coefficients of
                                            poly2.'
     0';'i';'j"                           Initialize indexes'
s1'  read'polyl'i"                        Read coefficient'
     for'i'step'l'until'63'rpeat'sl"
     stop'use's3"                         Too many coefficients'
s2'  i'i-'l';'nl"           i was incremented before discovery that
                            there is not another coefficient'
s4'  rdxit's6"
s5'  read'poly2'j"          for'j'step'l'until'63'rpeat's5"
     stop'use's3"           Too many coefficients'
s6'  j'i-'l';'n2"           prev'i+'nl';'lim i"          0';'i"
s7'  0';'sum"               i'i-'n2';'j"
     if'j'pos's8"           0';'j"
s8'  i';'lim j"             if'prev'i-'nl'neg's9"        nl';'lim j"
s9'  i'i-'j';'i-j"          sum'+'polyl'j'x'poly2'i-j';'sum"
     for'j'step'l'until'lim j'rpeat's9"                  sum';'prod'i"
     for'i'step'l'until'lim i'rpeat's7"                  0';'i"
s10' cr'1608'print'prod'i"               for'i'step'l'until'lim i'rpeat's10"
     cr'cr'use's3'"
```

## EXERCISE 20

("For" statement used) '

```
       dim'J'50"             index'n"
       iread'n"              0'i-'1';'-1"
s3'    read'J'n'1"           read'J'n"                 read'y"
       cr'2308'print'y"      cr'300'iprt'['n'i+'1']"   2008'print'J'n'1"
       cr'300'iprt'n"        2008'print'J'n"           n'i-'1';'n"
s2'    0'flo'['2'nx'n'i+'2']'/'y'x'J'n'1'-'J'n'2';'J'n"  cr'300'iprt'n"
       2008'print'J'n"       for'n'step'-1'until'0'rpeat's2"
       rdxit's4"             use's3"
s4'    stop"'
```

## EXERCISE 21

Matrix - Vector Product
n = number of rows in the vector and columns in the matrix, m = number of rows in the matrix'

```
       iread'n"                       iread'm"             dim'vctr'10'mtrx'101"
       index'k"                       dbind'ij"            m';'mtrx'0"
       1';'ij'0"                      0';'k"
s1'    k'i+'1';'k"                    if'k'i-'['n'i+'1']'zero's2"
       read'vctr'k"                   use's1"
s2'    1';'ij'1"
s3'    if'ij'1'i-'['n'i+'1']'zero's4"                      read'mtrx'ij"
       ij'1'i+'1';'ij'1"                                   use's3"
s4'    if'ij'0'i-'m'zero's5"          ij'0'i+'1';'ij'0"    use's2"
s5'    1';'ij'0"
s6'    0';'sum"                                            1';'ij'1';'k"
s7'    mtrx'ij'x'vctr'k'+'sum';'sum"                       k'i+'1';'k"
       if'k'i-'['n'i+'1']'zero's8"    ij'1'i+'1';'ij'1"    use's7"
s8'    cr'1608'print'sum"             if'ij'0'i-'m'zero's9"
       ij'0'i+'1';'ij'0"              use's6"
s9'    stop"'
```

## EXERCISE 22

```
sl'     iread'msize"               iread'rows"            iread'colms"
        if'225'i-'msize'neg'sl0"   dim'mtrx'226"
        dbind'ij"                  rows';'mtrx'0"         l';'ij'0"
s3'     l';'ij'l"
s4'     read'mtrx'ij"
        for'ij'l'step'l'until'colms'rpeat's4"
        for'ij'0'step'l'until'rows'rpeat's3"
        computation -----
sl0'    cr'daprt'uc2'M'lcl'a't'r'i'x' 't'o' 'l'a'r'g'e"
sll'    stop"'
```

## EXERCISE 23

```
        Matrix Product'
        iread'm"                              rows in A'
        iread'n"                              columns in A and rows in B'
        iread'k"                              columns in B'
        dim'mtrxa'401'mtrxb'401"              dbind'ij'jk"
        n';'mtrxa'0"                          k';'mtrxb'0"
        l';'ij'0"
sl'     l';'ij'l"
s2'     read'mtrxa'ij"                        for'ij'l'step'l'until'n'rpeat's2"
        for'ij'0'step'l'until'm'rpeat'sl"     l';'jk'0"
s3'     l';'jk'l"
s4'     read'mtrxb'jk"                        for'jk'l'step'l'until'k'rpeat's4"
        for'jk'0'step'l'until'n'rpeat's3"     l';'ij'0"
s5'     l';'jk'l"                             cr'0'i-'7';'crtn"
s6'     l';'ij'l';'jk'0"                      0';'sum"
s7'     mtrxa'ij'x'mtrxb'jk'+'sum';'sum"      jk'0'i+'l';'jk'0"
        for'ij'l'step'l'until'n'rpeat's7"     crtn'i+'l';'crtn"
        trn's8"                               cr'0'i-'7';'crtn"
s8'     1608'print'sum"                       for'jk'l'step'l'until'k'rpeat's6"
        for'ij'0'step'l'until'm'rpeat's5"     stop"'
```

## EXERCISE 24

```
        Binomial Coefficients'

        dim'coef'64"            index'm"            iread'nu"
        prev'i-'l';'limit"      0';'n-l"            .9999'99999'e'0';'coef'0"
s1'     0';'templ';'m"
s2'     templ';'temp2"          coef'm';'templ"     prev'+'temp2';'coef'm"
        for'm'step'l'until'n-l'rpeat's2"
        .9999'99999'e'0';'coef'm"           m is now n-l + l remark'
        for'n-l'step'l'until'limit'rpeat'sl"   0';'m"
s3'    cr'3000'iprt'['0'unflo'coef'm']"     for'm'step'l'until'nu'rpeat's3'"
```

## EXERCISE 25

```
        Factorial n'
        rdxit's4"
s1'     iread'n"                trn's3"
        if'n'i-'l'neg's3'zero's3"   .9999'99999'e'0';'factn"    l';'k"
s2'     factn'x'0'flo'k';'factn"    for'k'step'l'until'n'rpeat's2"
        cr'2008'print'factn"        use'sl"
s3'     cr'daprt'color'uc2'I'M'P'R'O'P'E'R' 'N'lcl'color"
s4'     stop'use'sl'"
```

## EXERCISE 26

```
s1'     compute w, u, and y -------
        if'y'neg's3'pos's2"                     ret's20'use's12"
        compute N(u) x N(w) -------             use'sl
s2'     set's4'to's5"
s3'     ret's20'use's10"
s4'     go to's0"
        compute Z(u) x N(w)--------             use'sl"
s5'     compute Z(w) x N(u)--------             use'sl"
s10'    Z of u and w computed and stored
s12'    N of u and w computed and stored
s20'    go to's0"
```

## EXERCISE 27

Each set of data is preceded by an integer for the iread j value, thereby indicating the statement to be used. In the first case 1 is iread and control goes to s10, second case, 2, and control goes to s30 and so on. In the fifth case where s20 is reused j is set again equal to 3. If j is set equal to 5 control is transferred to s101 and the program stops.'

```
        index'j"                  use's101'use's40'use's20'use's30'use's10"

s100'   iread'j"                  use's100'j"

s10'    read data, compute,
        print------               use's100"

s30'    read data, compute,
        print------               use's100"

s40'    If a variable amount of data is necessary in any of these subsections
        a rdxit may be set, as in

        rdxit's45"

        read data - data will be read in until a blankword is read, control
        is then transferred to s45.

s45'    compute, print----        use's100"

s101'   stop'"
```

## EXERCISE 28

limn: number of words in name code, limd: number of words in date code, run: run number, spl: space on first line, sp2: space on second line'

```
       iread'limn'iread'limd"     iread'run'iread'spl"              iread'sp2"

       dim'name'5'date'6"         index'j"                          1';'j';'page"

s1'    aread'name'j"              for'j'step'1'until'limn'rpeat's1"  1';'j"

s2'    aread'date"                for'j'step'1'until'limd'rpeat's2"  use's3"

s33'   page'i+'1';'page"

s3'    1';'j"

s4'    aprt'name'j"               for'j'step'1'until'limn'rpeat's4"

       spl'reprt' "               1';'j"

s5'    aprt'date'j"               for'j'step'1'until'limd'rpeat's5"

       1000'iprt'run"             sp2'reprt' "                      daprt'p'."

       1000'iprt'page"            64';'crtn"

s6'    Computation, and after cr' for blank line,

       crtn'i-'4';'crtn"          if'crtn'zero's33'pos's6"
```

At end of computation,

```
s7'   if'64'i-'crtn'zero's9"       4'reprt'cr4"
      crtn'i-'4';'crtn"            use's7"
s9'   run'i+'1';'run"              0';'page"                use's33'''
```

Name and date data to print, John Doe 4-July-62 Run No. +3'+6'+1'+71'+68'

```
uc Jlc o  h nspuc  Dlc o e 4 -uc J lc u l y  - 6 2cr uc Rlc u  nspuc Nlc o .sp
10640846'62320610'2f08464f'240a1064'08520j12'0a341420'101f0852'32061032'08462a06'
```