BASIC

Fourth Edition

A Manual for BASIC, the elementary algebraic language designed for use with the Dartmouth Time Sharing System.

John G. Kemeny
Thomas E. Kurtz

This edition prepared with the assistance of David Cochran.

1 January 1968

The development of the BASIC Language has been supported in part by the National Science Foundation under the terms of Grant NSF GE 3864.

Fourth Edition

Dartmouth College
Computation Center
1 January 1968

TABLE OF CONTENTS

INTRODUCTION

WHAT IS A PROGRAM?

A program is a set of directions, or a recipe, that is used to tell a computer how to provide an answer to some problem. It usually starts with the given data as the ingredients, contains a set of instructions to be performed or carried out in a certain order, and ends up with a set of answers as the cake. And, as with ordinary cakes, if you make a mistake in your program, you will end up with something else--perhaps hash!

Any program must fulfill two requirements before it can be carried out. The first is that it must be presented in a language that is understood by the "computer". If the program is a set of instructions for solving a system of linear equations and the "computer" is an English-speaking person, the program will be presented in some combination of mathematical notation and English. If the "computer" is a French-speaking person, the program must be in his language; and if the "computer" is a high-speed digital computer, the program must be presented in a language which the computer "understands".

The second requirement for all programs is that they must be completely and precisely stated. This requirement is crucial when dealing with a digital computer which has no ability to infer what you mean -- it does what you tell it to do, not what you meant to tell it.

We are, of course, talking about programs which provide

numerical answers to numerical problems. It is easy for a programmer to present a program in the English language, but such a program poses great difficulties for the computer because English is rich with ambiguities and redundancies, those qualities which make poetry possible but computing impossible. Instead, you present your program in a language which resembles ordinary mathematical notation, which has a simple vocabulary and grammar, and which permits a complete and precise specification of your program. The language you will use is BASIC which is, at the same time, precise, simple and easy to understand.

A first introduction to writing a BASIC program is given in Chapter I. This chapter includes all that you will need to know to write a wide variety of useful and interesting programs. Chapter II deals with more advanced computer techniques, and the Appendices contain a variety of reference materials.

Chapter I

A BASIC PRIMER

1.1 An Example

The following example is a complete BASIC program for solving a system of two simultaneous linear equations in two variables:

$$ax + by = c$$

$$dx + ey = f$$

and then solving two different systems, each differing from this system only in the constants c and f.

You should be able to solve this system, if ae - bd is not equal to 0, to find that

$$x = \frac{ce - bf}{ae - bd} \qquad \text{and} \qquad y = \frac{af - cd}{ae - bd}.$$

If ae - bd = 0, there is either no solution or there are infinitely many, but there is no unique solution. If you are rusty on solving such systems, take our word for it that this is correct. For now, we want you to understand the BASIC program for solving this system.

Study this example carefully--in most cases the purpose of each line in the program is self-evident--and then read the commentary and explanation.

```
10  READ A, B, D, E
15  LET G = A * E - B * D
20  IF G = 0 THEN 65
30  READ C, F
37  LET X = (C*E - B*F) / G
42  LET Y = (A*F - C*D) / G
55  PRINT X, Y
60  GØ TØ 30
65  PRINT "NØ UNIQUE SØLUTIØN"
70  DATA 1, 2, 4
80  DATA 2, -7, 5
85  DATA 1, 3, 4, -7
90  END
```

We immediately observe several things about this sample program. First, we see that the program uses only capital letters, since the teletype has only capital letters. And we see that the letter "oh" is distinguished from the numeral "zero" by having a diagonal slash through the "oh". We make the distinction since, in a computer program, it is not always possible to tell from the context whether the letter or the numeral was intended, unless they have a different appearance. This distinction is made automatically while typing, since the teletype has one key for "oh" and another for "zero"; and one key for "one", another for "eye", and no key for the lower case "el".

A second observation is that each line of the program begins with a number. These numbers are called line numbers and serve to identify the lines, each of which is called a statement. Thus, a program is made up of statements, most of which are instructions to the computer. Line numbers also serve to specify the order in which the statements are to be performed by the computer. This means that you may type your program in any order. Before the program is run, the computer sorts out and edits the program, putting the statements into

the order specified by their line numbers. (This editing process facilitates the correcting and changing of programs, as we shall explain later.)

A third observation is that each statement starts, after its line number, with an English word. This word denotes the type of the statement. There are several types of statements in BASIC, nine of which are discussed in this chapter. Seven of these nine appear in the sample program of this section.

A fourth observation, not at all obvious from the program, is that spaces have no significance in BASIC, except in messages which are to be printed out, as in line number 65 above. Thus, spaces may be used, or not used, at will to "pretty up" a program and make it more readable. Statement 10 could have been typed as 10READA,B,D,E and statement 15 as 15LETG=A*E-B*D.

With this preface, let us go through the example, step by step. The first statement, 10, is a READ statement. It must be accompanied by one or more DATA statements. When the computer encounters a READ statement while executing your program, it will cause the variables listed after the READ to be given values according to the next available numbers in the DATA statements. In the example, we read A in statement 10 and assign the value 1 to it from statement 70 and, similarly with B and 2, and with D and 4. At this point, we have exhausted the available data in statement 70, but there is more in statement 80, and we pick up from it the

the number 2 to be assigned to E.

We next go to statement 15, which is a LET statement, and first encounter a formula to be evaluated. (The asterisk "*" is obviously used to denote multiplication.) In this statement we direct the computer to compute the value of AE - BD, and to call the result G. In general, a LET statement directs the computer to set a variable equal to the formula on the right side of the equals sign. We know that if G is equal to zero, the system has no unique solution. Therefore, we next ask, in line 20, if G is equal to zero. If the computer discovers a "yes" answer to the question, it is directed to go to line 65, where it prints "NØ UNIQUE SØLUTION". From this point, it would go to the next statement. But lines 70, 80, and 85 give it no instructions, since DATA statements are not "executed", and it then goes to line 90 which tells it to "END" the program.

If the answer to the question "Is G equal to zero?" is "no", as it is in this example, the computer goes on to the next statement, in this case 30. (Thus, an IF-THEN tells the computer where to go if the "IF" condition is met, but to go on to the next statement if it is not met.) The computer is now directed to read the next two entries from the DATA statements, -7 and 5, (both are in statement 80) and to assign them to C and F respectively. The computer is now ready to solve the system

$$x + 2y = -7$$

$$4x + 2y = 5.$$

In statements 37 and 42, we direct the computer to compute the value of X and Y according to the formulas provided. Note that we must use parentheses to indicate that CE - BF is divided by G; without parentheses, only BF would be divided by G and the computer would let $X = CE - \frac{BF}{G}$.

The computer is told to print the two values computed, that of X and that of Y, in line 55. Having done this, it moves on to line 60 where it is directed back to line 30. If there are additional numbers in the DATA statements, as there are here in 85, the computer is told in line 30 to take the next one and assign it to C, and the one after that to F. Thus, the computer is now ready to solve the system

$$x + 2y = 1$$

$$4x + 2y = 3.$$

As before, it finds the solution in 37 and 42 and prints them out in 55, and then is directed in 60 to go back to 30.

In line 30 the computer reads two more values, 4 and -7, which it finds in line 85. It then proceeds to solve the system

$$x + 2y = 4$$

$$4x + 2y = -7$$

and to print out the solutions. It is directed back again to 30, but there are no more pairs of numbers available for C and F in the DATA statements. The computer then informs you that it is out of data, printing on the paper in your teletype "ØUT ØF DATA IN 30" and stops.

For a moment, let us look at the importance of the

various statements. For example, what would have happened if we had omitted line number 55? The answer is simple: the computer would have solved the three systems and then told us when it was out of data. However, since it was not asked to tell us (PRINT) its answers, it would not do it, and the solutions would be the computer's secret. What would have happened if we had left out line 20? In this problem just solved nothing would have happened. But, if G were equal to zero, we would have set the computer the impossible task of dividing by zero in 37 and 42, and it would tell us so emphatically, printing "DIVISIØN BY ZERØ IN 37" and "DIVISIØN BY ZERØ IN 42". Had we left out statement 60, the computer would have solved the first system, printed out the values of X and Y, and then gone on to line 65 where it would be directed to print "NØ UNIQUE SØLUTIØN". It would do this and then stop.

One very natural question arises from the seemingly arbitrary numbering of the statements: why this selection of line numbers? The answer is that the particular choice of line numbers <u>is</u> arbitrary, as long as the statements are numbered in the order which we want the machine to follow in executing the program. We could have numbered the statements 1, 2, 3, ..., 13, although we do not recommend this numbering. We would normally number the statements 10, 20, 30, ..., 130. We put the numbers such a distance apart so that we can later insert additional statements if we find that we have forgotten them in writing the program originally. Thus, if we find that

we have left out two statements between those numbered 40 and 50, we can give them any two numbers between 40 and 50-- say 44 and 46; and in the editing and sorting process, the computer will put them in their proper place.

Another question arises from the seemingly arbitrary placing of the elements of data in the DATA statements: why place them as they have been in the sample program? Here again the choice is arbitrary and we need only put the numbers in the order that we want them read (the first for A, the second for B, the third for D, the fourth for E, the fifth for C, the sixth for F, the seventh for the next C, etc.) In place of the three statements numbered 70, 80, and 85, we could have put

```
75 DATA 1, 2, 4, 2, -7, 5, 1, 3, 4, -7
```

or we could have written, perhaps more naturally,

```
70 DATA 1, 2, 4, 2
75 DATA -7, 5
80 DATA 1, 3
85 DATA 4, -7
```

to indicate that the coefficients appear in the first data statement and the various pairs of right-hand constants appear in the subsequent statements.

The program and the resulting run is shown below exactly as it appears on the teletype.

```
10 READ A, B, D, E
15 LET G = A * E - B * D
20 IF G = 0 THEN 65
30 READ C, F
37 LET X = ( C * E - B * F ) / G
42 LET Y = ( A * F - C * D ) / G
55 PRINT X, Y
60 GØ TØ 30
65 PRINT "NØ UNIQUE SØLUTIØN"
70 DATA 1, 2, 4
80 DATA 2, -7, 5
85 DATA 1, 3, 4, -7
90 END
RUN

 LINEAR        11:03        10/19/67


 4                -5.5
 0.666667         0.166667
-3.66667          3.83333
ØUT ØF DATA IN 30

TIME:     .10 SECS.
```

After typing the program, we type RUN followed by a carriage return. Up to this point the computer stores the program and does nothing with it. It is this command which directs the computer to execute your program.

Note that the computer, before printing out the answers, printed the name which we gave to the problem (LINEAR) and the time and date of the computation. At the end of the printed answers the machine tells us the amount of computing time used in our problem. The message "ØUT ØF DATA IN 30" here may be ignored. However, in some cases it indicates an error in the program: for more details see Sec. 1.7.2.

1.2 Formulas

The computer can perform a great many operations; it can add, subtract, multiply, divide, extract square roots, raise a number to a power, and find the sine of a number (or an angle measured in radians), etc.--and we shall now learn how to tell the computer to perform these various operations and to perform them in the order that we want them done.

The computer performs its primary function (that of computation) by evaluating formulas which are supplied in a program. These formulas are very similar to those used in standard mathematical calculation, with the exception that all BASIC formulas must be written on a single line. Five arithmetic operations can be used to write a formula, and these are listed in the following table:

Symbol	Example	Meaning
+	A + B	Addition (add B to A)
-	A - B	Subtraction (subtract B from A)
*	A * B	Multiplication (multiply B by A)
/	A / B	Division (divide A by B)
↑	X ↑ 2	Raise to the power (find X^2)

We must be careful with parentheses to make sure that we group together those things which we want together. We must also understand the order in which the computer does its work. For example, if we type A + B * C ↑ D, the computer will first raise C to the power D, multiply this result by B, and then add A to the resulting product. This is the same convention as is usual for A + B C^D. If this is not the order

intended, then we must use parentheses to indicate a different order. For example, if it is the product of B and C that we want raised to the power D, we must write A + (B * C)↑D; or, if we want to multiply A + B by C to the power D, we write (A + B) * C↑D. We could even add A to B, multiply their sum by C, and raise the product to the power D by writing ((A + B) * C)↑D. The order of priorities is summarized in the following rules:

1. The formula inside parentheses is computed before the parenthesized quantity is used in further computations.

2. In the absence of parentheses in a formula involving addition, multiplication, and the raising of a number to the power, the computer then performs the multiplication, and the addition comes last. Division has the same priority as multiplication, and subtraction the same as addition.

3. In the absence of parentheses in a formula involving only multiplication and division, the operations are performed from left to right, even as they are read. So also does the computer perform addition and subtraction from left to right.

These rules are illustrated in the previous example. The rules also tell us that the computer, faced with A - B - C, will (as usual) subtract B from A and then C from their difference; faced with A/B/C, it will divide A by B and <u>that</u> quotient by C. Given A↑B↑C, the computer will raise the number A to the power B and take the resulting number and raise it to the power C. If there is any question in your mind about the priority, put in more parentheses to eliminate possible ambiguities.

In addition to these five arithmetic operations, the

computer can evaluate several mathematical functions. These functions are given special 3-letter English names, as the following list shows:

Functions	Interpretation	
SIN (X)	Find the sine of X	X interpreted as a number, or as an angle measured in radians
CØS (X)	Find the cosine of X	
TAN (X)	Find the tangent of X	
CØT (X)	Find the cotangent of X	
ATN (X)	Find the arctangent of X	
EXP (X)	Find e^X	
LØG (X)	Find the natural logarithm of X (ln X)	
ABS (X)	Find the absolute value of X ($\|X\|$)	
SQR (X)	Find the square root of X ($\sqrt{X}$)	

Five other functions are also available in BASIC: INT, RND, SGN, NUM, and DET; these are reserved for explanation in Chapter II. In place of X, we may substitute any formula or any number in parentheses following any of these formulas. For example, we may ask the computer to find $\sqrt{4 + X^3}$ by writing SQR (4 + X↑3), or the arctangent of $3X - 2e^X + 8$ by writing ATN (3 * X - 2 * EXP (X) + 8).

If, sitting at the teletypewriter, you need the value of $(\frac{5}{6})^{17}$ you can write the two-line program

```
10  PRINT (5/6)↑17
20  END
```

and the computer will find the decimal form of this number and print it out in less time than it took you to type the program.

Since we have mentioned numbers and variables, we should be sure that we understand how to write numbers for the computer and what variables are allowed. A number may be positive or negative and it may contain up to nine digits, but it must be expressed in decimal form. For example, all of the following are numbers in BASIC: 2, -3.675, 123456789, -.987654321, and 483.4156. The following are not numbers in BASIC: 14/3, $\sqrt{7}$, and .00123456789. The first two are formulas but not numbers, and the last one has more than nine digits. We may ask the computer to find the decimal expansion of 14/3 or $\sqrt{7}$, and to do something with the resulting number, but we may not include either in a list of DATA. We gain further flexibility by use of the letter E, which stands for "times ten to the power". Thus, we may write .00123456789 in a form acceptable to the computer in any of several forms: .123456789E-2 or 123456789E-11 or 1234.56789E-6. We may write ten million as 1E7 (or 1E + 7) and 1965 as 1.965E3 (or 1.965E + 3). We do not write E7 as a number, but must write 1E7 to indicate that it is 1 that is multiplied by 10^7.

A numerical variable in BASIC is denoted by any letter, or by any letter followed by a single digit*. Thus, the computer will interpret E7 as a variable, along with A, X, N5, I0, and Ø1. A variable in BASIC stands for a number, usually one that is not known to the programmer at the time

* In this chapter we will discuss only numerical variables. See Section 2.7 for alphanumeric "string variables".

the program was written. Variables are given or assigned values by LET and READ statements. The value so assigned will not change until the next time a LET or READ statement is encountered with a value for that variable. However, all variables are set equal to 0 before a RUN. Thus, it is only necessary to assign a value to a variable when a value other than 0 is required.

Although the computer does little in the way of "correcting", during computation, it will sometimes help you when you forget to indicate absolute value. For example, if you ask for the square root of -7 or the logarithm of -5, the computer will give you the square root of 7 with the error message that you have asked for the square root of a negative number, or the logarithm of 5 with the error message that you have asked for the logarithm of a negative number.

Six other mathematical symbols are provided for in BASIC, symbols of relation, and these are used in IF-THEN statements where it is necessary to compare values. An example of the use of these relation symbols was given in the sample program in Section 1.

Any of the following six standard relations may be used:

Symbol	Example	Meaning
=	A = B	Is equal to (A is equal to B)
<	A < B	Is less than (A is less than B)
<=	A <=B	Is less than or equal to (A is less than or equal to B)
>	A > B	Is greater than (A is greater than B)
>=	A >=B	Is greater than or equal to (A is greater than or equal to B)
<>	A <>B	Is not equal to (A is not equal to B)

1.3 Loops

We are frequently interested in writing a program in which one or more portions are performed not just once but a number of times, perhaps with slight changes each time. In order to write the simplest program, the one in which this portion to be repeated is written just once, we use the programming device known as a loop.

The programs which use loops can, perhaps, be best illustrated and explained by two programs for the simple task of printing out a table of the first 100 positive integers together with the square root of each. Without a loop, our program would be 101 lines long and read:

```
10 PRINT 1, SQR (1)
20 PRINT 2, SQR (2)
30 PRINT 3, SQR (3)
   . . . . . .

990 PRINT 99, SQR (99)
1000 PRINT 100, SQR (100)
1010 END
```

With the following program, using one type of loop, we can obtain the same table with far fewer lines of instruction, 5 instead of 101:

```
10 LET X = 1
20 PRINT X, SQR (X)
30 LET X = X + 1
40 IF X <=  100 THEN 20
50 END
```

Statement 10 gives the value of 1 to X and "initializes" the loop. In line 20 both 1 and its square root are printed. Then, in line 30, X is increased by 1, to 2. Line 40 asks whether X is less than or equal to 100; an affirmative answer directs the computer back to line 20. Here it prints 2 and $\sqrt{2}$, and goes to 30. Again X is increased by 1, this time to 3, and at 40 it goes back to 20. This process is repeated--line 20 (print 3 and $\sqrt{3}$), line 30 (X = 4), line 40 (since $4 \leq 100$ go back to line 20), etc.--until the loop has been traversed 100 times. Then, after it has printed 100 and its square root, X becomes 101. The computer now receives a negative answer to the question in line 40 (X is <u>greater than</u> 100, not less than or equal to it), does not return to 20 but moves on to line 50, and ends the program. All loops contain four characteristics; initialization (line 10), the body (line 20), modification (line 30), and an exit test (line 40).

Because loops are so important and because loops of the type just illustrated arise so often, BASIC provides two statements to specify a loop even more simply. They are the FØR and NEXT statements, and their use is illustrated in the program:

```
10 FØR X = 1 TØ 100
20 PRINT X, SQR (X)
30 NEXT X
50 END
```

In line 10, X is set equal to 1, and a test is set up, like that of line 40 above. Line 30 carries out two tasks: X is increased by 1, and the test is carried out to determine whether to go back to 20 or go on. Thus lines 10 and 30 take the place of lines 10, 30, and 40 in the previous program-- and they are easier to use.

Note that the value of X is increased by 1 each time we go through the loop. If we wanted a different increase, we could specify it by writing

```
10 FØR X = 1 TØ 100 STEP 5
```

and the computer would assign 1 to X on the first time through the loop, 6 to X on the second time through, 11 on the third time, and 96 on the last time. Another step of 5 would take X beyond 100, so the program would proceed to the end after printing 96 and its square root. The STEP may be positive or negative, and we could have obtained the first table, printed in reverse order, by writing line 10 as

```
10 FØR X = 100 TØ 1 STEP -1
```

In the absence of a STEP instruction, a step size of +1 is assumed.

More complicated FØR statements are allowed. The initial value, the final value, and the step size may all be formulas of any complexity. For example, if N and Z have been specified earlier in the program, we could write

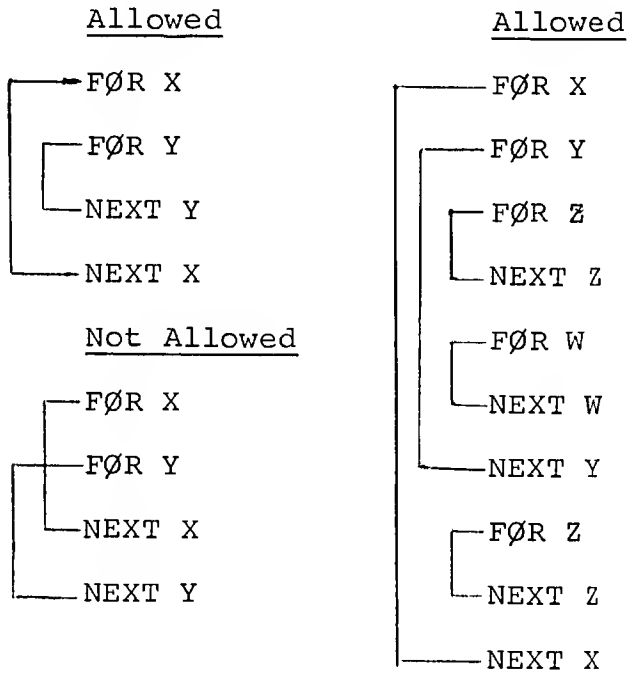
```
FØR  X = N + 7*Z TØ (Z-N)/3 STEP (N-4*Z)/10
```

For a positive step-size, the loop continues as long as the control variable is less than or equal to the final value. For a negative step-size, the loop continues as long as the control variable is greater than or equal to the final value.

If the initial value is greater than the final value (less than for negative step-size), then the body of the loop will not be performed at all, but the computer will immediately pass to the statement following the NEXT. As an example, the following program for adding up the first n integers will give the correct result 0 when n is 0.

```
10 READ  N
20 LET S = 0
30 FØR K = 1 TØ N
40 LET S = S + K
50 NEXT K
60 PRINT  S
70 GØ TØ 10
90 DATA 3, 10, 0
99 END
```

It is often useful to have loops within loops. These are called nested loops and can be expressed with FØR and NEXT statements. However, they must actually be nested and must not cross, as the following skeleton examples illustrate:



1.4 Lists and Tables

In addition to the ordinary variables used by BASIC, there are variables which can be used to designate the elements of a list or of a table. These are used where we might ordinarily use a subscript or a double subscript, for example the coefficients of a polynomial (a_0, a_1, a_2,...) or the elements of a matrix ($b_{i,j}$). The variables which we use in BASIC consist

of a single letter, which we call the name of the list, followed by the subscripts in parentheses. Thus, we might write A(0), A(1), A(2), etc. for the coefficients of the polynomial and B(1,1), B(1,2), etc. for the elements of the matrix.

We can enter the list A(0), A(1),...A(10) into a program very simply by the lines:

```
10 FØR I = 0 TØ 10
20 READ A(I)
30 NEXT I
40 DATA 2, 3, -5, 5, 2.2, 4, -9, 123, 4, -4, 3
```

We need no special instruction to the computer if no subscript greater than 10 occurs. However, if we want larger subscripts, we must use a DIM statement, to indicate to the computer that it has to save extra space for the list or table. When in doubt, indicate a larger dimension than you expect to use. For example, if we want a list of 15 numbers entered, we might write:

```
10 DIM A(25)
20 READ N
30 FØR I = 1 TØ N
40 READ A(I)
50 NEXT I
60 DATA 15
70 DATA 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47
```

Statements 20 and 60 could have been eliminated by writing 30 as FØR I = 1 TØ 15, but the form as typed would allow for the lengthening of the list by changing only statement 60, so long as it did not exceed 25.

We would enter a 3x5 table into a program by writing:

```
10 FOR I = 1 TO 3
20 FOR J = 1 TO 5
30 READ B (I,J)
40 NEXT J
50 NEXT I
60 DATA 2, 3, -5, -9, 2
70 DATA 4, -7, 3, 4, -2
80 DATA 3, -3, 5, 7, 8
```

Here again, we may enter a table with no dimension statement, and it will handle all the entries from B(0,0) to B(10,10). If you try to enter a table with a subscript greater than 10, without a DIM statement, you will get an error message telling you that you have a subscript error. This is easily rectified by entering the line:

```
5 DIM B(20,30)
```

if, for instance, we need a 20-by-30 table.

The single letter denoting a list or a table name may also be used to denote a simple variable without confusion. However, the same letter may not be used to denote both a list and a table in the same program. The form of the subscript is quite flexible, and you might have the list item B(I+K) or the table items B(I,K) or Q(A(3,7), B - C).

On the next page is a list and run of a problem which uses both a list and a table. The program computes the total sales of each of five salesmen, all of whom sell the same three products. The list P gives the price/item of the three products and the table S tells how many items of each product each man sold. You can see from the program that product no.1 sells for $1.25 per item, no. 2 for $4.30 per item, and no.3 for $2.50 per item; and also that salesman no. 1 sold 40 items of the first product, 10 of the second, and 35 of the third,

and so on. The program reads in the price list in lines 40-80, using data in lines 910-930. The same program could be used again, modifying only line 900 if the prices change, and only lines 910-930 to enter the sales in another month.

This sample program did not need a dimension statement, since the computer automatically saves enough space to allow all subscripts to run from 0 to 10. A DIM statement is normally used to save more space. But in a long program, requiring many small tables, DIM may be used to save less space for tables, in order to leave more for the program.

Since a DIM statement is not executed, it may be entered into the program on any line before END; it is convenient, however, to place DIM statements near the beginning of the program.

```
SALES1          11:05          10/20/67

10    FØR I = 1 TØ 3
20       READ P(I)
30    NEXT I
40    FØR I = 1 TØ 3
50       FØR J = 1 TØ 5
60          READ S(I,J)
70       NEXT J
80    NEXT I
90    FØR J = 1 TØ 5
100      LET S = 0
110      FØR I = 1 TØ 3
120         LET S = S + P(I)*S(I,J)
130      NEXT I
140      PRINT "TØTAL SALES FØR SALESMAN "J, "$" S
150   NEXT J
900   DATA 1.25, 4.30, 2.50
910   DATA 40, 20, 37, 29, 42
920   DATA 10, 16, 3, 21, 8
930   DATA 35, 47, 29, 16, 33
999   END
```

```
READY

RUN

 SALES1          11:06          10/20/67


TØTAL SALES FØR SALESMAN  1    $ 180.5
TØTAL SALES FØR SALESMAN  2    $ 211.3
TØTAL SALES FØR SALESMAN  3    $ 131.65
TØTAL SALES FØR SALESMAN  4    $ 166.55
TØTAL SALES FØR SALESMAN  5    $ 169.4

TIME:     .09 SECS.
```

1.5 Use of the Time-Sharing System

Now that we know something about writing a program in BASIC, how do we set about using a teletype to type in our program and then to have the computer solve our problem?

There are more details of the Time-Sharing System in Appendix A, but we shall learn enough in this section to handle a simple problem. To connect a teletype, which is equipped with a standard receiver on the right side, to the computer the following steps must be followed.

(1) lift receiver and wait for dial tone

(2) dial number (which should be posted prominently nearby)

(3) when ringing changes to high-pitched tone push the button labeled ORIG.

To connect a direct-line teletype which has no receiver it is only necessary to push the ORIG. button.

If the teletype is already connected and was being used by someone else you should type HELLØ and then push the key marked RETURN.

After any of the above procedures has been completed the computer will start typing and will then ask for your USER

NUMBER--. You are to type in your personal user number, for Dartmouth students the six digit student I. D. number, and again push the key marked RETURN. (You must, in fact, push the RETURN key after typing any line - only then does the line enter the computer.) Persons other than Dartmouth students can learn their user number from the Kiewit Computation Center.

The computer then types NEW ØR ØLD -- and you type the appropriate adjective: NEW if you are about to type a new problem and ØLD if you want to recover a problem on which you have been working earlier and have stored in the computer's memory.

The computer then asks NEW FILE NAME-- (or ØLD FILE NAME, as the case may be) and you type any combination of letters and digits you like*, but no more than eight. In the sample problem preceding you will remember that we named it SALES1. If you are recalling an old problem from the computer's memory, you must use exactly the same name as that which you gave the problem before you asked the computer to save it.

The computer then types READY and you should begin to type your program. Make sure that each line begins with a line number which contains no more than five digits and contains no spaces or non-digit characters. Also be sure to start at the very beginning of a line and to press the RETURN key at the completion of each line.

If, in the process of typing a statement, you make a typing error and notice it immediately, you can correct it by

* Some characters other than letters and digits are allowed in program names, but care is needed with certain ones. See Appendix D.

you can type STØP and the computation will cease. (If the teletype is actually typing, there is an express stop -- just press the "S" key.) It will then type READY and you can start to make your corrections. If you are in serious trouble, the combination CTRL - SHIFT - P will always give you a new start. (Hold down the CTRL and SHIFT keys with your left hand, and push P with your right hand.)

After you have all of the information you want, and are ready to leave the teletype, you should type GØØDBYE (or even BYE). The computer then types the time, and moves up your paper for ease in tearing off.

A sample use of the time-sharing system is shown below.

```
GE 600-LINE T/S FRØM DARTMØUTH
TERMINAL 138 ØN AT 11:13  10/20/67
USER NUMBER--123456
NEW ØR ØLD--NEW
NEW FILE NAME-- SAMPLE
READY
10 FØR N = 1 TØ 7
20 PRINT N,SQR(N)
30 NEXT N
40 PRINT "DØNE"
50 END
RUN

 SAMPLE        11:14        10/20/67


 1              1
 2              1.41421
 3              1.73205
 4              2
 5              2.23607
 6              2.44949
 7              2.64575
DØNE

TIME:     .07 SECS.

BYE
*** ØFF AT  11:14  HANØVER  10/20/67
```

1.6 Errors and "Debugging"

It may occasionally happen that the first run of a new problem will be free of errors and give the correct answers. But it is much more common that errors will be present and will have to be corrected. Errors are of two types: errors of form (or grammatical errors) which prevent the running of the program; and logical errors in the program which cause the computer to produce wrong answers or no answers at all.

Errors of form will cause error messages to be printed, and the various types of error messages are listed and explained in Sec. 2.8. Logical errors are often much harder to uncover, particularly when the program gives answers which seem to be nearly correct. In either case, after the errors are discovered, they can be corrected by changing lines, by inserting new lines, or by deleting lines from the program. As indicated in the last section, a line is changed by typing it correctly with the same line number; a line is inserted by typing it with a line number between those of two existing lines; and a line is deleted by typing its line number and pressing the RETURN key. Notice that you can insert a line only if the original line numbers are not consecutive integers. For this reason, most programmers will start out using line numbers that are multples of five or ten, but that is a matter of choice.

These corrections can be made at any time--whenever you notice them--either before or after a run. Since the computer sorts lines out and arranges them in order, a line may be retyped out of sequence. Simply retype the offending line with

its original line number.

As with most problems in computing, we can best illustrate the process of finding the errors (or "bugs") in a program, and correcting (or "debugging") it, by an example. Let us consider the problem of finding that value of X between 0 and 3 for which the sine of X is a maximum, and ask the machine to print out this value of X and the value of its sine. If you have studied trigonometry, you know that $\pi/2$ is the correct value; but we shall use the computer to test successive values of X from 0 to 3, first using intervals of .1, then of .01, and finally of .001. Thus, we shall ask the computer to find the sine of 0, of .1, of .2, of.3..., of 2.8, of 2.9, and of 3, and to determine which of these 31 values is the largest. It will do it by testing SIN(0) and SIN(.1) to see which is larger, and calling the larger of these two numbers M. Then it will pick the larger of M and SIN (.2) and call it M. This number will be checked against SIN (.3), and so on down the line. Each time a larger value of M is found, the value of X is "remembered" in X0. When it finishes, M will have been assigned to the largest value. It will then repeat the search, this time checking the 301 numbers 0, .01, .02, .03,..., 2.98, 2.99, and 3, finding the sine of each and checking to see which has the largest sine. At the end of each of these three searches, we want the computer to print three numbers: the value X0 which has the largest sine, the sine of that number, and the interval of search.

Before going to the teletype, we write a program and let

us assume that it is the following:

```
10 READ D
20 LET X0 = 0
30 FØR X = 0 TØ 3 STEP D
40 IF SIN (X)< = M THEN 100
50 LET X0 = X
60 LET M = SIN (X0)
70 PRINT X0, X, D
80 NEXT X0
90 GØ TØ 20
100 DATA .1, .01, .001
110 END
```

We shall list the entire sequence on the teletype and make explanatory comments on the right side.

```
NEW ØR ØLD--NEW
NEW FILE NAME-- MAXSIN
READY
10 READ D
20 LWR XO= 0
30 FØR X = 0 TØ 3 STEP D
40 IF SINE←(X) <= M THEN 100
50 LET XO=X
60 LET M = SIN(X)
70 PRINT XØ, X, D
80 NEXT Z←XO
90 GØ TØ 20
20 LET XO=0
100 DATA .1, .01, .001
110 END
RUN

 MAXSIN        11:35        10/20/67


ILLEGAL VARIABLE IN 70
NEXT WITHØUT FØR IN 80
FØR WITHØUT NEXT IN 30


TIME:     .07 SECS.

70 PRINT XO, X, D
40 IF SIN(X) <= M THEN 80
80 NEXT X
RUN

 MAXSIN        11:36        10/20/67


 0.1             0.1             0.1
 0.2             0.2             0.1
 0.3

TIME:     .10 SECS.
```

Notice the use of the backwards arrow to erase a character in line 40, which should have started IF SIN (X) etc., and in line 80.

After typing line 90, we notice that LET was mistyped in line 20, so we retype it, this time correctly.

After receiving the first error message, we inspect line 70 and find that we used XØ for a variable instead of X0. The next two error messages relate to lines 30 and 80, where we see that we mixed variables. This is corrected by changing line 80.

We make both of these changes by retyping lines 70 and 80. In looking over the program, we also notice that the IF-THEN statement in 40 directed the computer to a DATA statement and not to line 80 where it should go.

This is obviously incorrect. We are having every value of X printed, so we direct the machine to cease operations by typing S even while it is running. Note that the 'S' does not print. We ponder the program for a while, trying to figure out what is wrong with it. We notice that SIN(0) is compared with M on the first time through the loop, but we had assigned a value to X0 but not to M. However we recall that all variables are set equal to zero before a RUN so that line 20 is unneccessary.

```
20
RUN

 MAXSIN        11:37        10/20/67

 UNDEFINED LINE NUMBER 20 IN 90


 TIME:     .07 SECS.
```

Of course, line 90 sent us back to line 20 to repeat the operation and not back to line 10 to pick up a new value for D. We retype line 90 and then type RUN again.

```
 90 GØ TØ 10
RUN

 MAXSIN        11:43        10/20/67


 0.1              0.1              0.1
 0.2              0.2              0.1
 0.3

TIME:     .09 SECS.
```

We are about to print out the same table as before. It is printing out X0, the current value of X, and the interval size each time that it goes through the loop.

```
70
85 PRINT XO, M, D
5  PRINT "X VALUE", "SIN", RESØLUTIØN"
RUN

 MAXSIN        11:44        10/20/67


ILLEGAL VARIABLE IN 5


TIME:     .06 SECS.
```

We fix this by moving the PRINT statement outside the loop. Typing 70 deletes that line, and line 85 is outside of the loop. We also realize that we want M printed and not X. We also decide to put in headings for our columns by a PRINT statement.

There is an error in our PRINT statement: no left quotation mark for the third item.

Retype line 5, with all of the required quotation marks.

```
5  PRINT "X VALUE", "SINE", "RESØLUTIØN"
RUN

 MAXSIN        11:47        10/20/67


X VALUE          SINE              RESØLUTIØN
 1.6              0.999574          0.1
 1.57             1.                0.01
 1.57099          1.                0.001
ØUT ØF DATA IN 10


TIME:    1.68 SECS.
```

Exactly the desired results. Of the 31 numbers (0,.1,.2,.3, ...,2.8,2.9,3) it is 1.6 which has the largest sine, namely .999574. Similarly for finer subdivisions. The whole process took 1.68 seconds of the computer's time.

```
LIST

MAXSIN          11:48          10/20/67

5  PRINT "X VALUE", "SINE", "RESØLUTIØN"
10 READ D
30 FØR X = 0 TØ 3 STEP D
40 IF SIN(X) <= M THEN 80
50 LET XO=X
60 LET M = SIN(X)
80 NEXT X
85 PRINT XO, M, D
90 GØ TØ 10
100 DATA .1, .01, .001
110 END

READY

SAVE
READY
```

Having changed so many parts of the program, we ask for a list of the corrected program. Listing the corrected program, from time to time, is an important part of debugging.

The program is saved for later use. This should not be done unless future use is necessary.

In solving this problem, there is a common device which we did not use, namely the insertion of a PRINT statement when we wonder if the machine is computing what we think we asked it to compute. For example, if we wondered about M, we could have inserted 65 PRINT M, and we would have seen the values.

1.7 Summary of Elementary BASIC Statements

In this section we shall give a short and concise description of each of the types of BASIC statements discussed earlier in this chapter and add one statement to our list. In each form, we shall assume a line number, and shall use brackets to denote a general type. Thus, [variable] refers to any variable, which is a single letter, possibly followed by a single digit.

1.7.1 LET

This statement is not a statement of algebraic equality, but rather a command to the computer to perform certain computations and to assign the answer to a certain variable. Each LET statement is of the form: LET [variable] = [formula].

More generally several variables may be assigned the same value by a single LET statement.

Examples: (of the first type):

```
100  LET X = X + 1
259  LET W7 = (W-X4↑3)*(Z - A/(A - B) )- 17
```

(of the second type):

```
50  LET X = Y3 = A(3,1) = 1

90  LET W = Z = 3*X - 4*X↑2
```

1.7.2 READ and DATA

We use a READ statement to assign to the listed variables values obtained from a DATA statement. Neither statement is used without one of the other type. A READ statement causes the variables listed in it to be given, in order, the next available numbers in the collection of DATA statements. Before the program is run, the computer takes all of the DATA statements in the order in which they appear and creates a large data block. Each time a READ statement is encountered anywhere in the program, the data block supplies the next available number or numbers. If the data block runs out of data, with a READ statement still asking for more, the program is assumed to be done and we get an ØUT ØF DATA message.

Since we have to read in data before we can work with it, READ statements normally occur near the beginning of a program. The location of DATA statements is arbitrary, as long as they occur in the correct order. A common practice is to collect all DATA statements and place them just before the END state-

ment.

Each READ statement is of the form:

READ [sequence of variables] and each DATA statement of the form: DATA [sequence of numbers]

Examples:
```
150 READ X, Y, Z, X1, Y2, Q9
330 DATA 4, 2, 1.7
340 DATA 6.734E-3, -174.321, 3.14159265

234 READ B (K)
263 DATA 2, 3, 5, 7, 9, 11, 10, 8, 6, 4

 10 READ R (I,J)
440 DATA -3,5,-9,2.37,2.9876,-437.234E-5
450 DATA 2.765, 5.5576, 2.3789E2
```

Remember that only numbers are put in a DATA statement, and that 15/7 and $\sqrt{3}$ are formulas, not numbers.

1.7.3 PRINT

The PRINT statement has a number of different uses and is discussed in more detail in Chapter II. The common uses are (a) to print out the result of some computations, (b) to print out verbatim a message included in the program, (c) a combination of the two, and (d) to skip a line. We have seen examples of only the first two in our sample programs. Each type is slightly different in form, but all start with PRINT after the line number.

Examples of type (a):
```
100 PRINT X, SQR (X)
135 PRINT X, Y, Z, B*B - 4*A*C, EXP(A-B)
```

The first will print X and then, a few spaces to the right of that number, its square root. The second will print five different numbers:

$$X,\ Y,\ Z,\ B^2 - 4AC, \text{ and } e^{A-B}.$$

The computer will compute the two formulas and print them for you, as long as you have already given values to A, B, and C. It can print up to five numbers per line in this format.

Examples of type (b): 100 PRINT "NØ UNIQUE SØLUTIØN"
430 PRINT "X VALUE", "SINE", "RESØLUTIØN"

Both have been encountered in the sample programs. The first prints that simple statement; the second prints the three labels with spaces between them. The labels in 430 automatically line up with three numbers called for in a PRINT statement-- as seen in MAXSIN.

Examples of type (c): 150 PRINT "THE VALUE ØF X IS" X
30 PRINT "THE SQUARE RØØT ØF" X, "IS" SQR(X)

If the first has computed the value of X to be 3, it will print out: THE VALUE ØF X IS 3. If the second has computed the value of X to be 625, it will print out: THE SQUARE RØØT ØF 625 IS 25.

Example of type (d): 250 PRINT

The computer will advance the paper one line when it encounters this command.

1.7.4 GØ TØ

There are times in a program when you do not want all commands executed in the order that they appear in the program. An example of this occurs in the MAXSIN problem where the computer has computed X0, M, and D and printed them out in line 85. We did not want the program to go to the END statement yet, but to go through the same process for a different value of D. So we directed the computer to go back to line 10 with a GØ TØ statement. Each is in the form of GØ TØ [line number].

Example: 150 GØ TØ 75

1.7.5 IF -- THEN

There are times when we are interested in jumping the normal sequence of commands, if a certain relationship holds. For this we use an IF--THEN statement, sometimes called a conditional GØ TØ statement. Such a statement occurred at line 40 of MAXSIN. Each such statement is of the form

IF [formula] [relation] [formula] THEN [line number]

Examples: 40 IF SIN (X)<=M THEN 80
20 IF G = 0 THEN 65

The first asks if the sine of X is less than or equal to M, and directs the computer to skip to line 80 if it is. The second asks if G is equal to 0, and directs the computer to skip to line 65 if it is. In each case, if the answer to the question is No, the computer will go to the next line of the program.

1.7.6 ØN...GØ TØ

This command is the one which we have not encountered in the sample programs. The IF--THEN-- instruction allows a two-way fork in a program. ØN allows a many-way switch. For example:

```
ØN  X  GØ TØ 100, 200, 150
```

This causes the following:

If X = 1, the program goes to line 100,

If X = 2, the program goes to line 200,

If X = 3, the program goes to line 150.

More generally, in place of X any formula may occur, and there may be any number of line numbers in the instruction, as long as it fits on a single line. The value of the formula is computed and its integer part is taken. If this is 1, the program transfers to the line whose number is first on the list; if it is 2, to the second one, etc. If the integer part of the formula is below 1, or if it is larger than the number of line numbers listed, an error message is printed.

To increase the similarity between the ØN and IF-THEN instructions, the instruction

```
IF  X > 5 THEN 200
```

may also be written as

```
IF  X > 5 GØ TØ 200.
```

Conversely, "THEN" may be used in an "ØN" statement.

1.7.7 FØR and NEXT

We have already encountered the FØR and NEXT statements in our loops, and have seen that they go together, one at the entrance to the loop and one at the exit, directing the computer back to the entrance again. Every FØR statement is of the form

```
FØR [variable] = [formula] TØ [formula] STEP [formula]
```

Most commonly, the expressions will be integers and the STEP omitted. In the latter case, a step size of one is assumed. The accompanying NEXT statement is simple in form, but the variable must be precisely the same one as that following FØR in the FØR statement. Its form is NEXT [variable].

Examples:
```
30 FØR X = 0 TØ 3 STEP D
80 NEXT X

120 FØR X4 = (17 + CØS(Z))/3 TØ 3*SQR(10) STEP 1/4
235 NEXT X4

240 FØR X = 8 TØ 3 STEP -1

456 FØR J = -3 TØ 12 STEP 2
```

Notice that the step size may be a formula (1/4), a negative number (-1), or a positive number (2). In the example with lines 120 and 235, the successive values of X4 will be .25 apart, in increasing order. In the next example, the successive values of X will be 8, 7, 6, 5, 4, 3. In the last example, on successive trips through the loop, J will take on values -3, -1, 1, 3, 5, 7, 9, and 11.

If the initial, final, or step-size values are given as formulas, these formulas are evaluated once and for all upon entering the FØR statement. The control variable can be changed in the body of the loop; of course, the exit test always uses the latest value of this variable.

If you write 50 FØR Z = 2 TØ -2, without a negative step size, the body of the loop will not be performed and the computer will proceed to the statement immediately following the corresponding NEXT statement.

1.7.8 DIM

Whenever we want to enter a list or a table with a subscript greater than 10, we must use a DIM statement to inform the computer to save us sufficient room for the list or the table.

Examples:
```
20 DIM H(35)
35 DIM Q(5,25)
```

The first would enable us to enter a list of 35 items (or 36 if we use H(0)), and the latter a table 5 x 25, or by using row 0 and column 0 we get a 6 x 26 table.

1.7.9 END

Every program must have an END statement, and it must be the statement with the highest line number in the program. Its form is simple: a line number with END.

Example:
```
999 END
```

Chapter II

ADVANCED BASIC

2.1 More About PRINT

The uses of the PRINT statement were described in 1.7.3, but we shall give more detail here. Although the format of answers is automatically supplied for the beginner, the PRINT statement permits a greater flexibility for the more advanced programmer who wishes a different format for his output.

The teletype line is divided into five zones of fifteen spaces each. Some control of the use of these comes from the use of the comma: a comma is a signal to move to the next print zone or, if the fifth print zone has just been filled, to move to the first print zone of the next line.

More compact output can be obtained by use of the semi-colon. If a label (expression in quotes) is followed by a semi-colon, the label is printed with no space after it. If a variable is followed by a semi-colon, its value is printed in the following format:

First, a minus sign or a space (if it is positive),

then, the numerical value,

then, a single space.

Thus printing a list of numbers in semi-colon format will pack them in closest readable form.

For example, if you were to type the program

```
10 FØR I = 1 TØ 15
20 PRINT I
30 NEXT I
40 END
```

the teletype would print 1 at the beginning of a line, 2 at the

beginning of the next line, and so on, finally printing 15 on the fifteenth line. But, by changing line 20 to read

```
20 PRINT I,
```

you would have the numbers printed in the zones, reading

```
1          2          3          4          5
6          7          8          9          10
11         12         13         14         15
```

If you wanted the numbers printed in this fashion, but more tightly packed, you would change line 20 to replace the comma by a semi-colon:

```
20 PRINT I;
```

and the result would be printed

```
1  2  3  4  5  6  7  8  9  10  11  12  13  14  15
```

You should remember that a label inside quotation marks is printed just as it appears and also that the end of a PRINT signals a new line, unless a comma or semi-colon is the last symbol.

Thus, the instruction

```
50 PRINT X, Y
```

will result in the printing of two numbers and the return to the next line, while

```
50 PRINT X, Y,
```

will result in the printing of these two values and nc return--the next number to be printed will occur in the third zone, after the values of X and Y in the first two.

Since the end of a PRINT statement signals a new line, you will remember that

```
250 PRINT
```

will cause the typewriter to advance the paper one line. It will put a blank line in your program, if you want to use it for vertical spacing of your results, or it causes the completion of partially filled line, as illustrated in the following fragment of a program:

```
50 FØR M = 1 TØ N
110 FOR J = 0 TØ M
120 PRINT B(M,J);
130 NEXT J
140 PRINT
150 NEXT M
```

This program will print B(1,0) and next to it B(1,1). Without line 140, the teletype would then go on printing B(2,0), B(2,1), and B(2,2) on the same line, and then B(3,0), B(3,1) etc. Line 140 directs the teletype, after printing the B(1,1) value corresponding to M = 1, to start a new line and to do the same thing after printing the value of B(2,2) corresponding to M = 2, etc.

The instructions

```
50 PRINT "TIME-";"SHAR";"ING";
51 PRINT " AT";" DART";"MØUTH"
```

will result in the printing of

```
TIME-SHARING AT DARTMØUTH
```

Formatting of output can be controlled even further by use of the function TAB.

Insertion of TAB(17) will cause the teletype to move to column 17, just as if a tab had been set there. For this purpose the positions on a line are numbered from 0 through 74, and 75 is assumed to be the 0 position again.

More precisely, TAB may contain any formula as its argument. The value of the formula is computed, and its integer part is

taken. This in turn is treated modulo 75, to obtain a value from 0 through 74--as indicated above. The teletype is then moved forward to this position-- unless it has already passed this position, in which case the TAB is ignored.

For example, inserting the following line in a loop:

```
PRINT X;  TAB(12);  Y;  TAB(27);  Z
```

will cause the X-value to start in column 0, the Y-values in column 12 and the Z-values in column 27.

The following rules for the printing of numbers will help you in interpreting your printed results:

1. If a number is an integer, the decimal point is not printed. If the integer contains more than eight digits, the teletype will give you the first digit, followed by (a) a decimal point, (b) the next five digits, and (c) an E followed by the appropriate integer. For example, it will take 32,437,580,259 and write it as 3.24376E+10.

2. For any decimal number, no more than six significant digits are printed.

3. For a number less than 0.1, the E notation is used unless the entire significant part of the number can be printed as a six decimal number. Thus, 0.03456 means that the number is exactly .0345600000, while 3.45600E-2 means that the number has been rounded to .0345600.

4. Trailing zeros after the decimal point are not printed. The following program, in which we print out powers of 2, shows how numbers are printed.

```
10 FØR N = -5 TØ 30
20 PRINT 2↑N;
30 NEXT N
40 END

READY

RUN

 PØWERS        11:54        10/20/67


 0.03125  0.0625  0.125  0.25  0.5  1  2  4  8  16  32  64  128  256  512
 1024  2048  4096  8192  16384  32768  65536  131072  262144  524288
 1048576  2097152  4194304  8388608  16777216  33554432  67108864
 1.34218 E+8  2.68435 E+8  5.36871 E+8  1.07374 E+9

TIME:     .06 SECS.
```

2.2 Functions and DEF

Five functions were listed in Section 1.2 but not described. We will discuss INT, RND and SGN here and leave NUM and DET until the MAT section.

The INT function is the function which frequently appears in algebraic computation as [x], and it gives the greatest integer not greater than x. Thus INT(2.35) = 2, INT (-2.35)= -3, and INT (12) = 12.

One use of the INT function is to round numbers. We may use it to round to the nearest integer by asking for INT(X + .5). This will round 2.9, for example, to 3, by finding INT(2.9 + .5) = INT (3.4) = 3. You should convince yourself that this will indeed do the rounding guaranteed for it (it will round a number midway between two integers up to the larger of the integers).

It can also be used to round to any specific number of decimal places. For example, INT (10*X + .5)/10 ↑ 2 will round X

correct to two decimal places, and INT (X*10↑ D + .5)/10↑D round X correct to D decimal places.

The function RND produces a random number between 0 and 1. The form of RND does not require an argument.

If we want the first twenty random numbers, we write the program below and we get twenty six-digit decimals. This is illustrated in the following program.

```
10 FØR L = 1 TØ 20
20 PRINT RND,
30 NEXT L
40 END
RUN

 RNDNØS        13:24        10/20/67


 0.406533       0.88445       0.681969       0.939462       0.253358
 0.863799       0.880238      0.638311       0.602898       0.990032
 0.570427       0.897931      0.628126       0.613262       0.303217
 5.00548 E-2    0.393226      0.680219       0.632246       0.668218

TIME:     .08 SECS.

RUN

 RNDNØS        13:25        10/20/67


 0.406533       0.88445       0.681969       0.939462       0.253358
 0.863799

TIME:     .05 SECS.
```

Note that the second RUN was giving exactly the same "random" numbers as the first RUN. This greatly facilitates the debugging of programs that use the random number generator.

On the other hand, if we want twenty random one-digit integers,

we could change line 20 to read

```
20 PRINT INT(10*RND),
```

and we would then obtain

```
RNDNØS          13:26          10/20/67


 4              8              6              9              2
 8              8              6              6              9
 5              8              6              6              3
 0              3              6              6              6

TIME:     .07 SECS.
```

We can vary the type of random numbers we want. For example, if we want 20 random numbers ranging from 1 to 9 inclusive, we could change line 20 as shown

```
20 PRINT INT(9*RND + 1);
RUN

 RNDNØS          13:28          10/20/67


 4  8  7  9  3  8  8  6  6  9  6  9  6  6  3  1  4  7  6  7

TIME:     .07 SECS.
```

or we can obtain random numbers which are integers from 5 to 24 inclusive by changing line 20 as in the following example.

```
20 PRINT INT(20*RND + 5);
RUN

 RNDNØS          13:28          10/20/67


 13  22  18  23  10  22  22  17  17  24  16  22  17  17  11  6  12  18
 17  18

TIME:     .07 SECS.
```

In general, if we want our random numbers to be chosen from the A integers of which B is the smallest, we would call for INT(A*RND +B)

As we noted when we ran the first program of this section twice, we got the same numbers in the same order each time. However, we can get a different set by use of the instruction RANDØMIZE as in the following program. We show successive runs:

```
5 RANDØMIZE
10 FØR L = 1 TØ 20
20 PRINT INT(10*RND):
30 NEXT L
40 END
RUN

 RNDNØS        13:32        10/20/67

 1  9  4  2  1  1  6  6  3  8  4  9  8  6  5  8  6  2  6  0
TIME:     .07 SECS.
RUN

 RNDNØS        13:33        10/20/67

 1  1  4  6  6  6  0  5  3  8  4  0  8  1  0  5  1  8  0  1
TIME:     .04 SECS.
```

RANDØMIZE (or more briefly RANDØM) resets the random numbers in a random way. For example, if this is the first instruction in a program using random numbers, as in the above program, then repeated RUNs of the program will produce different results.

If the instruction is absent, then the "official list" of random numbers is obtained in the usual order.

It is suggested that a simulation model should be debugged without this instruction, so that one always obtains the same random numbers in test runs. After the program is debugged, one inserts

```
1  RANDØM
```

before starting production runs.

The SGN function is one which assigns the value 1 to any positive number, 0 to zero, and -1 to any negative number. Thus SGN(7.23) = 1, SGN(0)= 0, and SGN(-.2387) = -1. For example, the statement 50 ØN SGN(X) + 2 GØ TØ 100, 200, 300 will transfer to 100 if X $<$0, to 200 if X = 0, and to 300 if X $>$0.

In addition to the standard functions, you can define any other function which you expect to use a number of times in your program by use of a DEF statement. The name of the defined function must be three letters, the first two of which are FN. Hence, you may define up to 26 functions, e.g., FNA, FNB, etc.

The handiness of such a function can be seen in a program where you frequently need the function $e^{-x^2} + 5$. You would introduce the function by the line

```
30 DEF FNE(X) = EXP(-X↑2 + 5)
```

and later on call for various values of the function by FNE(.1), FNE(3.45),FNE(A+2), etc. Such definition can be a great time-saver when you want values of some function for a number of different values of the variable.

The DEF statement may occur anywhere in the program, and the expression to the right of the equal sign may be any formula which can be fitted onto one line. It may include any combination of other functions, including ones defined by different DEF statements, and it can involve other variables besides the ones denoting the argument of the function.

Each function defined may have zero, one, two, or more

variables. For example:

```
10 DEF FNB(X,Y) = 3*X*Y - Y↑3
105 DEF FNC(X,Y,Z,W) = FNB(X,Y)/FNB(Z,W)
530 DEF FNA = 3.1416*R↑2
```

In the definition of "FNA" the current value of R is used when FNA occurs. Similarly, if FNR is defined by

```
70 DEF FNR(X) = SQR(2 + LØG(X) - EXP(Y*Z)*(X + SIN(2*Z)))
```

and you have previously assigned values to Y and Z, you can ask for FNR(2.7). You can give new values to Y and Z before the next use of FNR.

The use of DEF as described so far is limited to those functions whose value may be computed within a single BASIC statement. However, there are functions which are tricky to define in one line even though it is possible to do so (e.g. the 'max' function), and many functions cannot be defined in one line. Thus the ability to have multiple line DEF's is extremely useful. We first illustrate the method with the above-mentioned example, the 'max' function. In this the possibility of using 'IF...THEN' as part of the definition is a great help:

```
10  DEF FNM(X,Y)
20  LET FNM = X
30  IF Y<= X THEN 50
40  LET FNM = Y
50  FNEND
```

The absence of the '=' sign in line 10 indicates that this is a multiple line DEF. FNEND in line 50 terminates the definition. The expression 'FNM' without an argument serves as a temporary

variable for the computation of the function value. The following example defines N-factorial:

```
10  DEF FNF(N)
20  LET FNF = 1
30  FØR K = 1 TØ N
40  LET FNF = K * FNF
50  NEXT K
60  FNEND
```

Any variable which is not an argument of FN_ in a DEF loop will have its current value in the program. Multiple line DEF's may not be nested, and there must not be a transfer from inside the DEF to outside its range, nor vice-versa.

2.3 GØSUB and RETURN

When a particular part of a program is to be performed more than one time, or possibly at several different places in the overall program, it is most efficiently programmed as a subroutine. The subroutine is entered with a GØSUB statement, where the number is the line number of the first statement in the subroutine. For example,

```
90  GØSUB 210
```

directs the computer to jump to line 210, the first line of the subroutine. The last line of the subroutine should be a return command directing the computer to return to the earlier part of the program. For example,

```
350 RETURN
```

will tell the computer to go back to the first line numbered greater than 90, and to continue the program there.

The following example, a program for determining the greatest common divisor of three integers using the Euclidean Algorithm, illustrates the use of a subroutine. The first two numbers are selected in lines 30 and 40 and their GCD is determined in the subroutine, lines 200-310. The GCD just found is called X in line 60, the third number is called Y in line 70, and the subroutine is entered from line 80 to find the GCD of these two numbers. This number is, of course, the greatest common divisor of the three given numbers and is printed out with them in line 90.

You may use a GØSUB inside a subroutine to perform yet another subroutine. This would be called "nested GØSUBs". In any case, it is absolutely necessary that a subroutine be left only with a RETURN statement, using a GØTØ or an IF-THEN to get out of a subroutine will not work properly. You may have several RETURNs in the subroutine so long as exactly one of them will be used.

```
10 PRINT " A", " B", " C", "GCD"
20 READ A,B,C
30 LET X = A
40 LET Y = B
50 GØSUB 200
60 LET X = G
70 LET Y = C
80 GØSUB 200
90 PRINT A,B,C,G
100 GØ TØ 20
110 DATA 60,90,120
120 DATA 38456,64872,98765
130 DATA 32,384,72
200 LET Q = INT(X/Y)
210 LET R = X - Q*Y
220 IF R = 0 THEN 300
230 LET X = Y
240 LET Y = R
250 GØ TØ 200
300 LET G = Y
310 RETURN
320 END
```

```
RUN

 GCD3NØS.      13:38        10/20/67

A              B              C              GCD
60             90             120             30
38456          64872          98765           1
32             384            72              8
ØUT ØF DATA IN 20

TIME:      .08 SECS.
```

2.4 INPUT

There are times when it is desirable to have data entered during the running of a program. This is particularly true when one person writes the program and enters it into the machine's memory*, and other persons are to supply the data. This may be done by an INPUT statement, which acts as a READ statement but does not draw numbers from a DATA statement. If, for example, you want the user to supply values for X and Y into a program, you will type

```
40 INPUT X, Y
```

before the first statement which is to use either of these numbers. When it encounters this statement, the computer will type a question mark. The user types two numbers, separated by a comma, presses the return key, and the computer goes on with the rest of the program.

Frequently an INPUT statement is combined with a PRINT statement to make sure that the user knows what the question mark is asking for. You might type

* See Appendix B

```
20 PRINT "YØUR VALUES ØF X, Y, AND Z ARE";
30 INPUT X, Y, Z
```

and the machine will type out

```
YØUR VALUES ØF X, Y, AND Z ARE?
```

Without the semicolon at the end of line 20, the question mark would have been printed on the next line.

Data entered via an INPUT statement is not saved with the program. Furthermore, it may take a long time to enter a large amount of data using INPUT. Therefore, INPUT should be used only when small amounts of data are to be entered, or when it is necessary to enter data during the running of the program such as with game-playing programs.

2.5 Some Miscellaneous Statements

Several other BASIC statements that may be useful from time to time are STØP, REM and RESTØRE.

STØP is entirely equivalent to GØTØ xxxxx, where xxxxx is the line number of the END statement in the program. It is useful in programs having more than one natural finishing point. For example, the following two program portions are exactly equivalent.

```
250  GØ TØ 999          250  STØP
     .........               ....
340  GØ TØ 999          340  STØP
     .........               ....
999  END                999  END
```

REM provides a means for inserting explanatory remarks in a program. The computer completely ignores the remainder of that line, allowing the programmer to follow the REM with directions

for using the program, with identifications of the parts of a long program, or with anything else that he wants. Although what follows REM is ignored, its line number may be used in a GØTØ or IF-THEN statement.

```
100  REM  INSERT DATA IN LINES 900-998.  THE FIRST
110  REM  NUMBER IS N, THE NUMBER ØF PØINTS.  THEN
120  REM  THE DATA PØINTS THEMSELVES ARE ENTERED, BY

200  REM  THIS IS A SUBRØUTINE FØR SØLVING EQUATIØNS

     .....

300  RETURN
     .....
520  GØSUB 200
```

There is a second method for adding comments to a program. Place an ' (apostrophe) at the end of the line, followed by a remark. Everything following the ' is ignored by BASIC. There is one obvious exception to this rule: if the line ended in a string (see Section 2.7), then BASIC will think that the apostrophe is part of the string, and the method will not work.

Sometimes it is necessary to use the data in a program more than once. The RESTØRE statement permits reading the data as many additional times as it is used. Whenever RESTØRE is encountered in a program, the computer restores the data block pointer to the first number. A subsequent READ statement will then start reading the data all over again. A word of warning-- if the desired data are preceded by code numbers or parameters, superfluous READ statements should be used to pass over these numbers. As an example, the following program portion reads the data, restores the data block to its original state, and reads the data

again. Note the use of line 570 to "pass over" the value of N, which is already known.

```
100   READ N
110   FØR I = 1 TØ N
120      READ X
      .....
200   NEXT I
      .....
560   RESTØRE
570   READ X
580   FØR I = 1 TØ N
590      READ X
      .....
```

2.6 MATRICES

Although you can work out for yourself programs which involve matrix computations, there is a special set of thirteen instructions for such computations. They are identified by the fact that each instruction must start with the word 'MAT'. They are

MAT READ A, B, C	Read the three matrices, their dimensions having been previously specified.
MAT C = ZER	Fill out C with zeroes.
MAT C = CØN	Fill out C with ones.
MAT C = IDN	Set up C as an identity matrix.
MAT PRINT A, B; C	Print the three matrices, with A and C in the regular format, but B closely packed.
MAT INPUT V	Calls for the input of a vector.
MAT B = A	Set the matrix B equal to the matrix A.
MAT C = A + B	Add the two matrices A and B.
MAT C = A - B	Subtract the matrix B from the matrix A.
MAT C = A*B	Multiply the matrix A by the number B.
MAT C = TRN(A)	Transpose the matrix A.

MAT C = (K)*A	Multiply the matrix A by the number K. The number K, which must be in parentheses, must also be given by a formula.
MAT C = INV(A)	Invert the matrix A.

These thirteen statements, with the addition of DIM, make matrix computations easier, and in combination with the ordinary BASIC instructions make the language much more powerful. However, the user has to be careful to keep (and to understand!) the conventions "built into" the language. We will discuss, below, the individual MAT statements.

The following convention has been adopted for MAT: while every vector has a component 0, and every matrix has a row 0 and a column 0, the MAT instructions ignore these. Thus if in a MAT instruction we have a matrix of dimension M-by-N, the rows are numbered 1, 2,...,M, and the columns 1, 2,...,N.

The DIM statement may simply indicate what the maximum dimension is to be. Thus, if we write

```
DIM M(20,35)
```

then M may have up to 20 rows and up to 35 columns. This statement is to save enough space for the matrix, and hence, the only care at this point is that the dimensions declared are large enough to accommodate the matrix. However, in the absence of DIM statements all vectors may have up to 10 components and matrices up to 10 rows and 10 columns. This is to say that in the absence of DIM statements this much space is automatically saved for vectors and matrices on their appearance in the program. The actual dimension of a matrix may be determined either when it is first set up (by a DIM statement) or when it is computed. Thus

```
10  DIM M(20,7)
    - - - -

50  MAT READ M
```

will read a 20-by-7 matrix for M, while

```
50  MAT READ M(17,30)
```

will read a 17-by-30 matrix for M, provided sufficient space has been saved for it by writing, for example,

```
10  DIM M(20,35).
```

The three instructions:

```
MAT  M = ZER

MAT  M = CØN

MAT  M = IDN,
```

which set up a matrix M with all components zero, all components equal to one, and as an identity matrix, respectively, act like MAT READ as far as the dimension of the resulting matrix is concerned. For example,

```
MAT  M = CØN(7,3)
```

sets up a 7-by-3 matrix with 1 in every component, while

```
MAT M = CØN
```

sets up a matrix, with ones in every component, and of dimension 10-by-10 unless previously dimensioned otherwise. It should be noted, however, that these instructions have no effect on row and column zero! Thus

```
10 DIM M(20,7)

20 MAT READ M(7,3)
.......
35 MAT M = CØN
.......
70 MAT M = ZER(15,7)
.......
90 MAT M = ZER(16,10)
```

will first read in a 7-by-3 matrix for M. Then it will set up a 7-by-3 matrix of all ones for M (the actual dimension having been set up as 7-by-3 in line 20.) Next it will set up M as a 15-by-7 all zero matrix. (Note that although this is larger than the previous M, it is within the limits set in 10.) But it will result in an error message in line 90. The limit set in line 10 is (20+1) x (7+1) = 168 components, and in 90 we are calling for (16+1) x (10+1) = 187 components. Thus, although the zero rows and columns are ignored in MAT instructions they play a role in determining dimension limits. So, for example

```
90  MAT M = ZER(25,5)
```

would not yield an error message.

It, perhaps, should be noted that an instruction such as MAT READ M(2,2) which sets up a matrix and which as we have said ignores the zero row and column does however affect the zero row and column. The redimensioning which may be implicit in an instruction causes the relocation of some numbers and so they may not appear subsequently in the same place. Thus even if we have first LET M(1,0) = M(2,0) = 1, say, and then MAT READ M(2,2) the values of M(1,0) and M(2,0) will now be 0. Thus, when using MAT instructions, it is best not to use row and column zero.

The instruction

```
MAT PRINT A , B ; C
```

will cause the three matrices to be printed with A and C in the normal format (i.e. with five components to a line and starting each new row on a new line) and B closely packed.

Vectors may be used in place of matrices, as long as the above rules are observed. Since a vector like V(I) is treated

as a column vector by BASIC, a row vector has to be introduced as a matrix that has only one row. namely row 1. Thus

```
DIM X(7), Y(0,5)
```

introduces a 7-component column vector and a 5-component row vector.

If V is a vector then

```
MAT PRINT V
```

will print the vector V as a column vector.

```
MAT PRINT V,
```

will print V as a row vector, five numbers to the line, while

```
MAT PRINT V;
```

will print V as a row vector, closely packed.

The instruction

```
MAT INPUT V
```

will call for the input of a vector. The number of components in the vector need not be specified! Normally the input is limited by having to be typed on one line. However by ending the line of input with & (before carriage return) the machine will ask for more input on the next line. Note that, although the number of components need not be specified, if we wish to input more than 10 numbers we must save sufficient space with a DIM statement. After the input the function NUM will equal the number of components and V(1), V(2), ..., V(NUM) will be the numbers inputted. This allows variable length input. For example

```
5 LET  S = 0
10 MAT INPUT V
20 LET N = NUM
30 IF N = 0 THEN 99
40 FØR  I = 1 TØ N
45 LET S = S + V(I)
50 NEXT I
60 PRINT S/N
70 GØ TØ  5
99 END
```

allows the user to type in sets of numbers, which are averaged. The program takes advantage of the fact that zero numbers may be inputted, and uses this as a signal to stop. Thus, the user can stop by simply pushing "carriage return" on an input request.

MAT B = A

This sets B up to be the same as A and in doing so dimensions B to be the same as A, provided sufficient space has been saved for B.

MAT C = A + B, MAT C = A - B

For these to be legal A and B must have the same dimensions, and enough space must be saved for C. They cause C to assume the same dimensions as A and B. Instructions MAT A = A $\pm$ B are legal - the indicated operation is performed and the answer stored in A. Only a single arithmetic operation is allowed so MAT D = A + B - C is illegal but may be achieved with two MAT instructions.

MAT C = A * B

For this to be legal it is necessary that the number of columns in A be equal to the number of rows in B. For example, if A has dimension L-by-M and B has dimension M-by-N then C = A * B will have dimension L-by-N. It should be noted that while MAT A = A + B may be legal, MAT A = A * B will result in nonsense! There is good reason for this. In adding two matrices we may immediately store the answer in one of the matrices; but if we attempt to do this in multiplying two matrices, we will destroy components which would be needed to complete the computation!! MAT B = A * A is, of course, legal provided A is a 'square' matrix.

MAT C = TRN(A)

This lets C be the transpose of the matrix A. Thus if A is an M-by-N matrix C will be an N-by-M matrix.

MAT C = (K) * A

This lets C be the matrix A multiplied by the number K (i.e. each component of A is multiplied by K to form the components of C). The number K, which must be in parentheses, may be replaced by a formula. MAT A = (K) * A is legal.

MAT C = INV(A)

This lets C be the inverse of A. (A must, of course, be a 'square' matrix.) The function DET is available after the execution of the inversion, and will equal the determinant of A. This enables the user to decide whether the determinant was large enough for the inverse to be meaningful. In particular, attempting to invert a singular matrix will not cause the program

to stop, but DET is set equal to 0. Of course, the user may actually want the determinant of a matrix; he may obtain this by inverting the matrix and then seeing what value DET has.

We close this section with two illustrations of matrix programs. The first one reads in A and B in line 30 and in so doing sets up the correct dimensions. Then, in line 40, A + A is computed and the answer is called C - this automatically dimensions C to be the same as A. Note that the data in line 90 results in A being 2-by-3 and B being 3-by-3. Both MAT PRINT formats are illustrated, and one method of labeling a matrix print is shown.

```
MATRIX          13:48          10/20/67

10 DIM A(20,20),B(20,20),C(20,20)
20 READ M,N
30 MAT READ A(M,N),B(N,N)
40 MAT C = A + A
50 MAT PRINT C;
60 MAT C = A*B
70 PRINT
75 PRINT "A*B =",
80 MAT PRINT C
90 DATA 2,3
91 DATA 1,2,3
92 DATA 4,5,6
93 DATA 1,0,-1
94 DATA 0,-1,-1
95 DATA -1,0,0
99 END

READY

RUN

 MATRIX          13:48          10/20/67


 2   4   6
 8   10  12

A*B =
-2              -2              -3
-2              -5              -9

TIME:     .07 SECS.
```

The second example inverts an N-by-N Hilbert Matrix

1	1/2	1/3	. . .	1/N
1/2	1/3	1/4	. . .	1/N+1
1/3	1/4	1/5	. . .	1/N+2
.	.	.	. . .	
.	.	.	. . .	
.	.	.	. . .	
1/N	1/N+1	1/N+2		1/2N-1

Ordinary BASIC instructions are used to set up the matrix in lines 50 to 90. Note that this occurs after correct dimensions have been declared. Then a single instruction results in the computation of the inverse, and one more instruction prints it. The fact that the function DET is available after an inversion is also taken advantage of in line 130 to print the value of the determinant of A. In this example we have supplied 4 for N in the DATA statement and have made a run for this case.

```
HILMAT        13:52        10/20/67

5  REM THIS PRØGRAM INVERTS AN N-BY-N HILBERT MATRIX
10 DIM A(20,20),B(20,20)
20 READ N
30 MAT A = CØN(N,N)
50 FØR I = 1 TØ N
60 FØR J = 1 TØ N
70 LET A(I,J) = 1/(I+J-1)
80 NEXT J
90 NEXT I
100 MAT B = INV(A)
115 PRINT "INV(A) =",
120 MAT PRINT B;
125 PRINT
130 PRINT "DETERMINANT ØF A =" DET
190 DATA 4
199 END

READY
```

```
RUN

 HILMAT        13:52        10/20/67


INV(A) =
 16.0001 -120.001  240.003 -140.002
-120.001  1200.01 -2700.03  1680.02
 240.003 -2700.03  6480.08 -4200.05
-140.002  1680.02 -4200.05  2800.03

DETERMINANT ØF A = 1.65342 E-7

TIME:     .08 SECS.
```

It may be of interest that a well-behaved 20 x 20 matrix is inverted in about .5 seconds. However, the reader is warned that beyond N = 7 the Hilbert matrix cannot be inverted because of severe round-off errors.

Although it is not possible to create N-dimensionsal arrays in BASIC, the method outline below will simulate them. The example is of a 3-dimensional array but it has been written in such a way that it could be changed to 4 or higher dimensions easily. We use the fact that functions can have any number of variables and we set up a one-to-one correspondence between the components of the array and the components of a vector. The number of components in the vector will equal the product of the dimensions of the array. For example, if the array has dimensions 2, 3, 5 then the vector will have 30 components. A multiple line DEF could be used in place of the simple DEF in line 30 if the user wished to include error messages. The print-out is in the form of two 3 x 5 matrices.

```
10 DIM V(1000)
20 MAT READ D(3)
30 DEF FNA(I,J,K) = ((I-1)*D(2) +(J-1))*D(3) + K
50 FØR I = 1 TØ D(1)
60 FØR J = 1 TØ D(2)
70 FØR K = 1 TØ D(3)
80 LET V(FNA(I,J,K)) = I+2*J+K↑2
90 PRINT V(FNA(I,J,K)),
100 NEXT K
110 NEXT J
112 PRINT
115 PRINT
120 NEXT I
900 DATA 2,3,5
999 END
RUN

 3-ARRAY       08:07        10/27/67

 4              7            12           19           28
 6              9            14           21           30
 8              11           16           23           32


 5              8            13           20           29
 7              10           15           22           31
 9              12           17           24           33
```

2.7 Alphanumeric Information

Our discussion of BASIC in previous sections dealt only with numerical information. However, BASIC will also handle alphabetic or combined alphabetic-numeric information. We define a string to be a sequence of characters, each of which is either a letter, a digit, a space, or some other printable character (but not a quotation mark for reasons which will become obvious below).

We may introduce variables for single strings and we may introduce 'string' vectors (but not 'string' matrices). Any ordinary variable followed by a $ (dollar sign) will stand for a string. For example A$ or C7$ can denote strings. A vector variable followed by $, e.g. V$( ), will denote a list of strings. Thus, V$(7) is the 7th string in the list.

First of all strings may be read and printed. For example,

```
10 READ A$, B$, C$
20 PRINT C$; B$; A$
30 DATA ING, SHAR, TIME-
40 END
```

will print the word "time-sharing". Note that the effect of the semi-colon in the print statement is consistent with that discussed in the section on PRINT, i.e. with alphanumeric output the semi-colon causes close packing whether that output is in quotes or is the value of a variable. Commas and TAB's may be used as in any other PRINT statement. The loop

```
70  FØR I = 1 TØ 12
80  READ M$(I)
90  NEXT I
```

will read a list of 12 strings.

In place of the READ and PRINT corresponding MAT instructions may be used for lists. For example, MAT PRINT M$; will cause the members of the list to be printed without spaces between them. We may also use INPUT or MAT INPUT. After a MAT INPUT the function NUM will equal the number of strings inputted.

As usual, lists are assumed to have no more than 10 elements, otherwise a DIM statement is required. A statement like

```
10  DIM  M$(20)
```

saves room for twenty strings in the M$-list.

In the DATA statements numbers and strings may be intermixed. Numbers will be assigned only to numerical variables,

and strings only to string-variables. Strings in DATA statements are recognized by the fact that they start with a letter. If a string does not start with a letter, it must be enclosed in quotes. The same requirement holds for a string containing a comma. For example:

```
90 DATA 10, ABC, 5, "4FG", "SEPT. 22, 1967", 2
```

The only convention on INPUT is that a string containing a comma must be enclosed in quotes.

With a MAT INPUT a string containing a comma or an ampersand must be enclosed in quotes. For example:

```
"MR. & MRS. SMITH", MR. JONES
```

is in correct format for a response to a MAT INPUT.

In any of the three ways of getting string information into a program, DATA, INPUT or MAT INPUT leading blanks are ignored unless the string, including the blanks, is enclosed in quotes.

If in doubt about the use of strings, it is safest to enclose the string in quotes.

Strings (in quotes) or string variables may occur in LET and IF-THEN statements. The following two examples are self-explanatory:

```
10 LET Y$ = "YES"
20 IF Z7$ = "YES" THEN 200
```

The relation "<" is interpreted as "earlier in alphabetic order". This also serves to define the other relations. In any comparison trailing blanks in a string are ignored. Thus

"YES" = "YES ".

We illustrate these possibilities by the following program which reads a list of strings, and alphabetizes them:

```
10  DIM L$(50)
20  READ N
30  MAT READ L$(N)
40  FØR I = 1 TØ N
50  FØR J = 1 TØ N-I
60  IF L$(J)  = L$(J+1) THEN 100
70  LET A$ = L$(J)
80  LET L$(J) = L$(J+1)
90  LET L$(J+1) = A$
100  NEXT J
110  NEXT I
120  MAT PRINT L$
900  DATA 5, ØNE, TWØ, THREE, FØUR, FIVE
999  END
```

If we omit the $ signs in this program, it serves to read a list of numbers and prints them in increasing order.

A rather common use is illustrated by the following:

```
330  PRINT "DØ YØU WISH TØ CØNTINUE";
340  INPUT A$
350  IF A$ = "YES" THEN 10
360  STØP
```

Numeric and string DATA are kept in two separate blocks, and these act independently of each other. Therefore RESTØRE will restore both the numerical data and the string data. RESTØRE* will restore only the numerical data and RESTORE$ will

restore only the string data.

In BASIC it is very easy to "get at" the individual digits in a number by using the function INT. It is possible to "get at" the individual characters in a string with the instruction CHANGE. The use of CHANGE is best illustrated with examples.

```
5 DIM A(65)
10 READ A$
15 CHANGE A$ TØ A
20 FØR I = 0 TØ A(0)
25 PRINT A(I);
30 NEXT I
40 DATA ABCDEFGHIJKLMNØPQRSTUVWXYZ
45 END

READY
RUN

 CHANGE        13:55        10/20/67


 26   65   66   67   68   69   70   71   72   73   74   75   76   77   78   79   80
 82   83   84   85   86   87   88   89   90

TIME:      .06 SECS.
```

The instruction **CHANGE A$ TØ A** in line 15 has caused the vector A to have as its zero component the number of character in the string A$ and to have certain numbers in the other components. These numbers are the BASIC 'code' numbers for the characters appearing in the string e.g. A(1) is 65 - the BASIC code number for A.

The BASIC code for the printable characters is:

Character	BASIC Code No.
" "	32
"!"	33
""	34
"#"	35
"$"	36
"%"	37
"&"	38
"'"	39
"("	40
")"	41
"*"	42
"+"	43
","	44
"-"	45
"."	46
"/"	47
"0"	48
"1"	49
"2"	50
"3"	51
"4"	52
"5"	53
"6"	54
"7"	55
"8"	56
"9"	57
":"	58
";"	59
"<"	60
"="	61
">"	62
"?"	63
"@"	64
"A"	65
"B"	66
"C"	67
"D"	68
"E"	69
"F"	70
"G"	71
"H"	72
"I"	73
"J"	74
"K"	75
"L"	76
"M"	77
"N"	78
"Ø"	79
"P"	80
"Q"	81
"R"	82
"S"	83

Character	BASIC Code No.
"T"	84
"U"	85
"V"	86
"W"	87
"X"	88
"Y"	89
"Z"	90
"["	91
"\"	92
"]"	93
"↑"	94

Additional symbols useful on output are:

← (backward arrow)	95
EØT (end of transmission)	4
BELL (rings bell in teletype)	7
LF (line feed)	10
CR (carriage return)	13
RUB-ØUT (tape use only)	127

This is not a complete list - there are 128 characters numbered 0 through 127. Some of these numbers duplicate the above (on some teletypes) some are for teletypes with upper and lower case letters, and some are useless.

The other use of CHANGE is illustrated now:

```
10  FOR I = 0 TØ 5
15  READ  A(I)
20  NEXT  I
25  DATA 5, 65, 66, 67, 68, 69
30  CHANGE  A TØ A$
35  PRINT A$
40  END
```

This will print ABCDE because the numbers 65 - 69 are the code numbers for A - E. Before CHANGE is used in the 'vector to string' direction we must give the number of characters which are to be in the string as the zero component of the vector. Above A(0) is read as 5. A final example:

```
5 DIM V(128)
10 PRINT "WHAT DØ YØU WANT THE VECTØR V TØ BE";
20 MAT INPUT V
30 LET V(0) = NUM
40 CHANGE V TØ A$
50 PRINT A$
60 GØ TØ 10
70 END

READY

RUN

 EXAMPLE       13:59       10/20/67


WHAT DØ YØU WANT THE VECTØR V TØ BE? 40,32,45,60,45,89,90
( -<-YZ
WHAT DØ YØU WANT THE VECTØR V TØ BE? 32,33,34,35,36,37,38,39,40,41,42,43
? 44,45,46,47,48,49,50
 !"#$%&'()*+,-./012
WHAT DØ YØU WANT THE VECTØR V TØ BE? 4
```

Note that in this example we have used the availability of the function NUM after a MAT INPUT to find the number of characters in the string which is to result from line 40. Giving the input 4 on request gets the response EØT (end of transmission) which turns off the teletype.

2.8 Error Messages

The various error messages that can occur in BASIC, together with their interpretation, are now given:

Error Message	Interpretation
DIMENSIØN TØØ LARGE	The size of a list or table is too large for the available storage. Make them smaller. (See Appendix B.)
ILLEGAL CØNSTANT	More than nine digits or incorrect form in a constant number, or a number out of bounds (>1.70141 E + 38).
ILLEGAL FØRMULA	Perhaps the most common error message, may indicate missing parentheses, illegal variable names, missing multiply signs, illegal numbers, or many other errors. Check the statement thoroughly.
ILLEGAL RELATIØN	Something is wrong with the relational expression in an IF-THEN statement. Check to see if you used one of the six permissible relational symbols.
ILLEGAL LINE NUMBER	Line number is of incorrect form, or contains more than five digits.
ILLEGAL INSTRUCTIØN	Other than one of the legal BASIC instructions has been used following the line number.
ILLEGAL VARIABLE	An illegal variable name has been used.
ILLEGAL FØRMAT	The format of an instruction is wrong. See especially IF - THEN's and FØR's.

END IS NØT LAST	Self-explanatory, it also occurs if there are two or more END statements in the program.
NØ END INSTRUCTIØN	The program has no END statement.
NØ NUMERIC DATA	There is at least one READ statement calling for numeric data, but no numeric data, in the program.
NØ STRING DATA	There is at least one READ statement calling for string data, but no string data, in the program.
UNDEFINED FUNCTION FN_	A function such as FNF( ) has been used without appearing in a DEF statement. Check for typographical errors.
UNDEFINED LINE NUMBER	The line number appearing in an ØN, GØ TØ or IF THEN statement does not appear as a line number in the program.
NEXT WITHOUT FØR	An incorrect NEXT statement, perhaps with a wrong variable given.
FØR WITHØUT NEXT	A missing NEXT statement - can occur in conjunction with the previous one.
ILLEGAL CHARACTER	The line contains a character foreign to BASIC.
INCØRRECT NUMBER ØF ARGUMENTS	A function has been defined with a certain number of arguments and later called for with a different number.
CUT PRØGRAM ØR DIMS	Either the program is too long, or the amount of space reserved by the DIM statements is too much, or a combination of these. This message can be eliminated by either cutting the length of the program, or by reducing the size of the lists and tables, reducing the length of printed labels, or reducing the number of simple variables.
INCØRRECT NUMBER ØF SUBSCRIPTS	A matrix with one subscript or a vector with two.
FØRS NESTED TØØ DEEPLY	Too many FØR statements before the necessary NEXT statements.
ILLEGAL LINE REFERENCE	There is some character other than a number in this transfer (e.g. GØ TØ) statement where the line number should be.

UNFINISHED DEF	A multiple line DEF has not been ended correctly with FNEND.
NESTED DEF	Multiple line DEF's must not be nested. (However, functions defined elsewhere may be used in a DEF.)
EXPRESSIØN TØØ CØMPLICATED	Probable that too many parentheses have been used. Use two statements in place of this one.
TØØ MANY CØNSTANTS	Too many constants which the machine finds difficult to handle in binary. Put some in as DATA.
ILLEGAL MAT TRANSPØSE	MAT A = TRN(A) is illegal.
ILLEGAL MAT FUNCTIØN	Other than one of the legal MAT functions listed in Chapter 2 has been used.
ILLEGAL MAT MULTIPLE	Self explanatory. MAT A = A * B is illegal.
MISMATCHED STRING ØPERATIØN	The machine has been asked to combine two strings algebraically, to compare a string and a number, or to assign a number to a string variable or vice versa.
SYSTEM ERRØR	Error in BASIC!! Please report.

The following error messages can occur after your program has run for awhile. Thus, they may conceivably occur after the first part of your answers have been printed. All of these errors indicate the line number in which the error occurred.

ØUT ØF DATA	A READ statement for which there is no DATA has been encountered. This may mean a normal end of your program, and should be ignored in those cases. Otherwise, it means that you haven't supplied enough DATA. In either case, the program stops.

SUBSCRIPT ERRØR	A subscript has been called for that lies outside the range specified in the DIM statement, or if no DIM statement applies, outside the range 0 through 10. The program stops.
RETURN BEFØRE GØSUB	Occurs if a RETURN is encountered before a corresponding GØSUB during the running of a program. (Note: BASIC does not require the GØSUB to have an earlier statement number--only to perform a GØSUB before performing a RETURN.) The program stops.
GØSUB NESTED TØØ DEEPLY	Too many GØSUBS without a RETURN. It may mean that subroutines are being exited by GØTØ or IF-THEN statements rather than by RETURNs. The program stops.
DIVISIØN BY ZERØ	A division by zero has been attempted. The computer assumes the answer is + ∞ (for which the largest positive number, about 1.7E+38, is used) and continues running the program.
ZERØ TØ A NEGATIVE PØWER	A computation of the form 0 ↑ (-1) has been attempted. The computer procedes as in the previous case.
DIMENSIØN ERRØR	A dimension inconsistency has occurred in connection with one of the MAT statements. The program stops.
ABSØLUTE VALUE RAISED TØ PØWER	A computation of the form (-3)↑ 2.7 has been attempted. The computer supplies (ABS(-3)) ↑ 2.7 and continues. Note: (-3) ↑ 3 is correctly computed to give -27.
ØVERFLØW	A number larger than about 1.70141 E+38 has been generated. The computer supplies + (or-) ∞ and continues running the program.
UNDERFLØW	A number in absolute size smaller than about 1.46937E-39 has been generated. The computer supplies 0 and continues running the program. In many circumstances, underflow is permissible and may be ignored.
EXP TØØ LARGE	The argument of the exponential function is ≥ 88.029 + ∞ is supplied for the value of the exponential, and the running is continued.

LØG ØF NEGATIVE NUMBER	The program has attempted to calculate the logarithm of a negative number. The computer supplies the logarithm of the absolute value and continues.
LØG ØF ZERØ	The program has attempted to calculate the logarithm of 0. The computer supplies - ∞ and continues running the program.
SQUARE RØØT ØF A NEGATIVE NUMBER	The program has attempted to extract the square root of a negative number. The computer supplies the square root of the absolute value and continues running the program.
ØN EVALUATED ØUT ØF RANGE	The integer part of the variable in the ØN instruction is less than one or greater than the number of line numbers listed. The program stops.
USELESS LØØP IN	The program contains a loop which does nothing and keeps looping. The program stops.
ØUT ØF RØØM	The amount of space reserved for the program was not sufficient. Try adding a string DIM statement to acquire more space, e.g. DIM A$(1000).
TIME UP -- LØØPING IN	Occurs only with TEACH programs and means that the program is taking an unreasonable length of RUN time to do the task assigned.

INPUT DATA NØT IN CØRRECT FØRMAT -- RETYPE IT Self-explanatory.

NØT ENØUGH INPUT -- ADD MØRE Self-explanatory.

TØØ MUCH INPUT -- EXCESS IGNØRED Self-explanatory.

2.9 Limitations on BASIC

While the language BASIC has no inherent limitations on the size or complexity of programs, its implementation on any specific computer is limited by the physical characteristics of the machine and the nature of the implementation. The following

remarks are applicable to the current implementation on a GE-635 time-sharing system. Even here, they are subject to change as the implementation is improved.

Since there is only a finite amount of space available in the memory of the 635, there is a limit to the size of BASIC programs that may be RUN. However, these limitations do not apply to any single feature of the program, but to its total requirements. While this optimizes the use of space, it makes it difficult to give a precise rule. The following is a useful rule of thumb:

Let C = no. of characters in program

M = no. of components in all vectors and matrices

S = no. of strings

then $C/4 + M + S < 8000$ is a requirement. The user will know M and S, and may obtain C by a LENGTH command. But this rule only assures that the program will be compiled (translated from BASIC to machine language). If the program generates an unusually long code, or if there are many long strings, then it may still fail to RUN. This results in an ØUT ØF RØØM message. The user must then cut the code, or reduce the size of matrices or of strings.

The same message may, however, occur for an entirely different reason, even in a short program. Since the length of strings is not known at the start, BASIC may not save enough room for very long strings. Thus, if the user plans to use strings of more than 60 characters, it is best to add a vacuous DIM A$(1000) statement, to assure sufficient space.

There is one additional limitation: Only 100 constants may occur in the program. However, certain simple constants--such as small integers -- do not count towards this quota.

APPENDICES

Appendix A -- Using the Time-Sharing System

The Dartmouth Time-Sharing System consists of a large central computer with a number of input-output stations (currently, models 33 and 35 teletype machines). Individuals using the input-output stations are able to "share" the use of the computer with each other in such a way as to suggest that each has sole use of the computer. The teletype machines are the devices through which the user communicates with the computer.

THE KEYBOARD

The teletype keyboard is a standard typewriter keyboard for the most part. There are some special keys that the user must be familiar with.

RETURN — This is located at the right-hand end of the third row of keys, and does more than act as an ordinary carriage return. The computer ignores the line being typed until this key is pushed.

This key is located on the "oh" key when either SHIFT key is pressed. It is used to delete the character of space immediately preceding the "←". If this key is pressed N times, the characters or spaces in the N preceding spaces will be deleted.

ABCWT←←DE appears as ABCDE when RETURN is pushed.
AB C←←CDE appears as ABCDE when RETURN is pushed.

CTRL - X — The CTRL (control) key is located at the left-hand end of the second row of keys beside A. The combination CTRL-X acts as a carriage return but also deletes the entire line being typed.

CTRL-SHIFT-P — This combination will cause the machine to stop whatever it is doing, e.g. if in a RUN, it will stop and give the RUN time.

(Some languages available on the time-sharing system use the three characters "\", "[", and "]". They are located on the keys "L", "K", and "M" respectively when either SHIFT key is pushed.)

TELETYPE OPERATION ...

...FROM DIRECT LINE TELETYPES.

Besides the keyboard itself there are 4 buttons necessary to operate the machine.

BUTTON	LOCATION	FUNCTION
ORIG	leftmost of six small buttons on the right.	turns on the teletype
CLR	next to ORIG	turns off teletype
LOC LF	left of the space bar on model 35 teletypes only.	feeds paper to permit tearing off.
BUZ-RLS	rightmost of six small buttons on the right	turns off buzzer, which signals low paper supply.

If the teletype is on a direct line to the computer, pushing the ORIG button is all that is necessary to connect up with the computer. To disconnect from the computer, type GØØDBYE or BYE. If that fails, push CLR.

...FROM "LONG" DISTANCE TELETYPES:

Some teletypes utilize telephone company lines through dataphone facilities to make connection with the computer. Besides the buttons described above, the following buttons have additional functions.

BUTTON	LOCATION	FUNCTION
TEL	above telephone dial.	secures dial tone for the teletypes on a call-up basis.
CLR	next to ORIG	breaks telephone connection as well as disconnects the teletype from the computer.

In order to connect up with the computer from long distance teletypes, follow this routine:

1. Push the TEL button and wait for dial tone.

2. Dial one of the dataphones at the Computation Center.

3. When you hear a high-pitched tone, push the ORIG button.

In order to disconnect from a long distance teletype, type GØØDBYE or BYE. If that fails, push CLR.

REQUIRED STATEMENTS AT SIGN-ON

Once the teletype is connected to the computer it will start typing. Remember that all typed lines must be followed by a carriage return (RETURN). The machine will ask for certain information which you will supply by typing the information when asked for it, and following each response with a carriage return.

First, it asks for the user's number, which is either the six-digit student ID number, or a special number assigned to the user by the Computation Center. Then it will ask whether it is a new or old program you will be working on. A new program is one which the user is about to start on, while an old program has been saved in memory for future use.
Finally, it will ask for the new or old problem name. After the machine types READY the user may begin with his new program or pick up where he left off on his old program. A typical sequence follows. (The underline indicates information typed by the user.)

USER NUMBER -- 999999

NEW OR OLD -- NEW

NEW PROBLEM NAME -- M36-2

READY.

CONTROL COMMANDS

There are a number of commands that may be given to the computer by typing the command at the start of a new line (no line number) and following the command with a carriage return (RETURN).

COMMAND	MEANING
CATALØG	The computer types a list of the names of all programs currently being saved by that user.

EDIT	See Appendix C
HELLØ	To be typed when a new user takes over on a teletype that is already connected.
LENGTH	Gives the user the length of the program by printing the number of characters. (Cf. Section 2.9)
LIST	Causes an up-to-date listing of the program to be typed out.
LIST--XXXXX	Causes an up-to-date listing of the program to be typed out beginning at line number XXXXX and continuing to the end.
LISTNH	Causes a listing of the program to be typed out without a heading.
LISTNHXXXXX	Causes a listing from line XXXXX to be typed without a heading.
NEW	Erases the program currently being worked on and asks for a NEW PRØBLEM NAME.
ØLD	Erases the program currently being worked on and asks for an ØLD PRØBLEM NAME.
RENAME	Permits you to change the problem name of the program currently being worked on, but does not destroy the program.
REPLACE	Saves the current program and in doing so erases a saved program of the same name from memory.
RUN	Begins the computation of a program.
SAVE	Saves the program intact for later use. (To retrieve saved programs, type ØLD).
SCRATCH	Destroys the problem currently being worked on, but leaves the user number and problem name intact. It gives the user a "clean sheet" to work on.

STATUS	Gives an indication of the status of the teletype machine you are using (running, edit, idle).
STØP	Stops the computation at once. It can be typed even when the teletype is typing at full speed.
SYSTEM	Permits the user to change systems (BASIC, ALGØL, etc.)
TTY	Supplies the following information: Teletype number, user number, language being used, program being used, and status of teletype.
UNSAVE	Erases a saved program from memory. The memory of the computer is finite and this command should be used to free space in memory for programs of other users.

TELETYPES WITH PAPER TAPE ATTACHMENTS

A program may be saved on paper tape very simply. Just type LISTNH, and turn on the paper tape unit. This will output the program on tape in a format suitable for reading into the teletype at a later date. To reinput such a tape, two special commands are needed. After typing NEW, and specifying the name of the file, type TAPE, and turn on the tape unit. The program will then be read into the computer. After turning off the tape unit, type KEY to return control to the keyboard.

Appendix B -- Library Files.

Programs which have been written and which are of interest or use to others may be saved permanently in the computer in such a way that they become available to all users. Programs in this category are called library programs and are accessed as in the following example:

```
NEW ØR ØLD -- ØLD

ØLD FILE NAME -- BASICT***
```

The program name is BASICT and the three *'s inform the computer to search for the file in the library. The library programs are grouped by subject and a list of these groups may be obtained by listing the library program DARTCAT***. More detailed documentation on library programs may be obtained

from the librarian of the Computation Center.

Additions are made to the library from time to time; thus programs which have been written and which are suitable for inclusion in the library should be communicated to the Kiewit Computation Center.

Appendix C -- EDIT

There are several EDIT commands available to the user. These enable the user to edit a program in certain ways or to combine several programs in a number of different ways. The EDIT commands available are listed with a short description of each. Detailed information on EDIT commands is available from the librarian of the Computation Center. This documentation also indicates the current status of these commands.

EDIT RESEQUENCE	Renumbers the lines in the program.
EDIT PAGE	Lists the program in a paged format.
EDIT DELETE	Deletes lines or blocks of lines from the program.
EDIT EXTRACT	Extracts lines or blocks of lines from the program.
EDIT LIST	Lists certain lines or blocks of lines of the program.
EDIT WEAVE	Weaves two or more (up to nine) programs together, keeping line numbers intact.
EDIT APPEND	Appends a program at the end of another one.
EDIT MERGE	Inserts a program into another (main) program at a specified line number and resequences the whole if necessary.
EDIT MØVE	Moves lines or blocks of lines to new positions in the program.
EDIT TRACE	Adds certain print statements to the program and causes a RUN to be performed.
EDIT RUNØFF	Produces a neat paginated copy of textual material without line numbers.

For an explanation of a given command, type - for example - EDIT EXPLAIN RESEQUENCE.

Appendix D -- Program Names

As noted in Appendix B three *'s at the end of a program name indicate that you want a library program. Thus asterisks should be avoided in program names.

The EDIT commands make use of commas and semi-colons and may use / in their format (See Appendix C). For this reason programs which have these symbols in their name will not EDIT successfully.

The following simple rule will provide safe names. In naming a program, do not use the symbols: * , ; / $.

Appendix E -- Future Plans

At the time of writing many new features to improve the system and make it even more useful are being developed. When these features become available they will be fully documented and a manual containing details will be published. A brief description of each new feature follows:

(1.) Passwords

It is intended that users will be able to protect their catalog with a password. In fact it will be impossible to get "into" the machine using a user number protected by a password without supplying the password on request.

Also available will be passwords for individual programs. Thus if a program has been given a password it will be impossible to access the program without supplying the password. Giving the command CATALØG will still only cause a list of the names, and not the passwords, of the programs saved, to be typed out.

(2.) Background BASIC

BASIC users having long-running programs, large amounts of data, or large amounts of output will be able to request a background run of their problem. Background BASIC will provide for using the card-reader for input of data and the high speed printer for outputting results, as well as use of tapes.

(3.) Data Files

Data files will allow users to save results of computations inside the computer.

(i) Numerical data files will consist of a list of

numbers (in binary notation). It is not intended that these files be created directly from the teletype but by means of a WRITE instruction in BASIC.

BASIC will access these files by means of READ instructions and so may be listed through a user program using PRINT.

It will be necessary to declare the files to be read with a FILES command in BASIC before any use is made of the files. This will establish the files to be used initially and will save space for them. It will be possible to keep track of the number of numbers read from a file and to keep track of how many numbers there are in the file. It will also be possible to RESTØRE all the data files being used or just one at a time if required. In fact, it will also be possible to RESTØRE a file to a particular position rather than to the beginning of the file.

(ii) Data files will also be avilable for strings - it will be required that the name of a string file end with $ as a protection for the user. Most of the comments above are applicable to string files.

(iii) It will also be possible to create teletype-format files. The purpose of these is to be able to save DATA separately from the program, but to be able to list or modify it from the teletype. BASIC will treat these files as if they were INPUT from a teletype.

It should be pointed out that teletype files will be somewhat more difficult to manipulate than numeric or string files. However, they will allow mixed numerical and string data, and it will be possible to list or modify them directly from a teletype.

(4.) Chaining of Programs

This feature will allow one program to initiate a RUN on another program without the two programs communicating directly. However, the first program could write into a data file which the second will read.

(5.) Segmenting Programs

This feature will allow the writing of a long program in several independent segments, which will fully communicate, and which may be debugged separately. It will also allow multiple use of segments, e.g. the use of a library sub-routine as a segment.

The advantages of this will be better service from the machine, since small programs can get scheduled sooner, and easier debugging since it is much easier to debug a short program than a long one.

(6.) Multiple Teletype Connections

It will be possible for one teletype to be the "master" and to control several other "slave" teletypes to a certain extent. The slave teletypes will be limited to being able to supply data in response to an INPUT. The "master" teletype will be the only one able to give system commands. The linkage will be set up via user numbers and will require a positive action from all the terminals concerned.

This feature will allow controlled experiments with a small group of subjects, cooperation on solving problems, and competitive games.

(7.) Saving of Compiled Programs

At the present time saved programs are re-compiled every time they are RUN. This is wasteful of machine time and so it is intended that the instructions

```
ØLD

ØLD FILE NAME -- ABC

READY

CØMPILE GHI
```

will cause ABC to be compiled, but not executed. If the compilation is successful the resulting machine language program will be called GHI and will become the current program of the user. It will then be possible to save this version of the program. The resulting program can be RUN, but not LISTED.