IBM

# The Use of APL in Teaching

This paper was originally published by Queen's University, Kingston, Ontario, under the title "The Role of Computers in Teaching". It appeared as No. 13 in Queen's papers on Pure and Applied Mathematics.

The programming systems, APL/360 and APL/1130 are available from the program information department of IBM through IBM representatives.

These programs and their documentation have been contributed to the Program Information Department by an IBM employee and are provided by the IBM Corporation as part of its service to customers. The programs and their documentation are essentially in the author's original form and have not been subjected to any formal testing. IBM makes no warranty expressed or implied as to the documentation, function, or performance of these programs and the user of the programs is expected to make the final evaluation as to the usefulness of the programs in his own environment. There is no committed maintenance for the programs.

Questions concerning the use of the programs should be directed to the author. Any changes to the programs will be announced in the appropriate Catalog of Programs; however, the changes will not be distributed automatically to users. When such an announcement occurs, users should order only the material (documentation, machine readable or both) as indicated in the appropriate Catalog of Programs.

# F O R E W O R D

The present work consists of a summary of eight lectures delivered by Dr. Kenneth Iverson at Queen's University on March 21 and 22, 1968, to an enthusiastic audience of professors and High School teachers from Ontario and Quebec. His lively informal lecturing style, his directness, his evident profound and comprehensive knowledge of computing and its applications greatly impressed all his hearers.

I heard only one criticism. His lectures were not sufficiently philosophical; they contained no sweeping generalizations! This criticism was conveyed to him during the course of his lectures but had essentially no effect! I gained the impression that this avoidance of any of the many glowing glib assessments of computers and their possibilities, which are so common in popular literature, is a deliberate pedagogical device on the part of Dr. Iverson. Certainly he is enthusiastic about the computer and its possibilities for teaching. But if it is to prove of real use, and not provide merely time-and-money-wasting gilt to the educational process, the computer must be approached with a down-to-earth, salty attitude.

His aim of showing precisely what we can reasonably expect to accomplish in teaching with the computer dictated, Dr. Iverson's approach. So in what follows, the reader will find a series of concrete examples of what has been done, which, taken as a whole, define the immediate future potentialities of the computer in teaching as Dr. Iverson sees them at present.

In order to present meaningful examples, some notation is needed. For this purpose, Dr. Iverson used the programming language, APL, which had its origin in his work at Harvard. APL is a simple, consistent variation and generalization of ordinary algebraic notation. Many of my colleagues regard APL as the simplest and at the same time the most powerful programming language suitable for conversational mode which is currently available. The footnote on p. 3 describes the development of the APL system.

APL has been used extensively at several Canadian Universities, including Queen's, with a group of faculty and honours students. The reader who reads the first few pages of this book attentively will quickly pick up this new language, as did most of Dr. Iverson's auditors, and then easily appreciate the significance of the examples which he uses to illustrate the four areas of teaching in which the computer can immediately be put to effective use.

A. J. Coleman,
Head, Department of Mathematics
Queen's University

TABLE OF CONTENTS

## INTRODUCTION

The computer is fast becoming an important tool in teaching at all levels. It is important that teachers begin to learn its use, both to exploit its present possibilities and to explore and develop further uses.

The immediate potential of computers in teaching is due to two rather recent developments. The first is time-sharing, which permits a single central computer to serve a large number of relatively inexpensive typewriter terminals connected by ordinary phone lines. The second is the simplification of programming, which permits a teacher or student to make effective use of a computer without devoting time to the study of inessential details.

The computer can serve many purposes in education, including the scheduling of classes and classrooms and the performance of other purely administrative record-keeping. The present discussion will be limited to those purposes directly related to teaching. The following aspects of computer use will be treated in turn: 1) exercises and experimentation, 2) examinations, 3) drill, 4) computer programs as the framework of a discipline.

Mathematics appears as the most likely area for the application of computers, and indeed most of the examples in the present paper are drawn from algebra. However, other quantitative topics such as physics, chemistry, and statistics are equally amenable. In non-quantitative topics such as language and history, the immediate promise lies primarily in the categories of drill and examinations.

Before proceeding to discuss any one of the uses of computers in teaching, it will be necessary to consider the techniques of communicating with a computer. This topic is called programming.

## Programming

A time-shared computer is used via a terminal, which behaves like an ordinary typewriter except that each keystroke is transmitted (encoded as a set of tones) to the computer over telephone lines, and that responses from the computer are typed automatically. The following is a typical discourse via such a terminal:

```
      3.4+5.6
9
      3.4×7
2.38
      X←3
      Y←5
      X×Y
15
      X+Y
8
      (X+Y)×(X-Y)
¯16
```

The first line of the discourse is entered by the user. After he strikes the carriage return the computer sends its response 9 (which is then typed automatically), followed by a carriage return and five spaces. The spaces automatically indent the user's entry so as to distinguish entries from responses. The entries on lines 5 and 6 assign values to the names (commonly referred to as variables) $X$ and $Y$. The succeeding lines show how expressions involving such variables are evaluated, i.e., by substituting for each variable the value previously assigned to it.

It is important that the communication with the computer be unambiguous, and for that reason the rules of discourse are more rigid than might be inferred from the preceding example. For example, entering

```
      3.4 PLUS 5.6
```

will result in the typing of an error message rather than the sum of 3.4 and 5.6. Both the basic operations permitted (such as +, ×, ÷ and *) and the rules for combining them (for example, $(X+Y)\times(X-Y)$) are limited in number and precisely defined. For this reason, the discourse is said to be formal and the "language" defined by these rules of discourse is called a formal language.

There are hundreds of formal programming languages in existence, but there is only one well-established formal language which students must learn in any case, namely, the language embodied in algebraic notation. The present discussion will be couched in terms of a formal programming language* which is very similar to algebraic notation. Each of the examples used will show the actual discourse carried out on a terminal.

In order to show the relation to algebraic notation and to clarify the reasons for departures therefrom, certain examples will first be stated in the familiar notation. Consider the following well-known method for computing the area of a triangle with sides of length $A$,$B$, and $C$: calculate the semi-perimeter, take the product of the semi-perimeter and the difference between it and each of the three sides, then raise this product to the one-half power (i.e., take its square root).

In algebraic notation, the method is stated as follows:

$$P=A+B+C$$
$$S=P\div 2$$
$$(S(S-A)(S-B)(S-C))^{1/2}$$

*The APL language was first defined by K. E. Iverson in A Programming Language (Wiley, 1962) and was later developed in collaboration with A. D. Falkoff. The APL Terminal System was designed with the additional collaboration of L. M. Breed, who, with R. D. Moore, also designed the S/360 implementation. The system was programmed for S/360 by Breed, Moore, and R. H. Lathwell, with assistance from L. Woodrum. The present implementation also benefitted from experience with an earlier version, programmed for the IBM 7090 by Breed and P. S. Abrams.

Other relevant publications are:

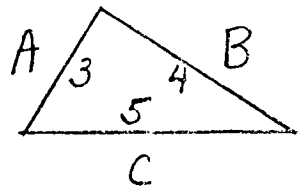
Pakin, S., APL\360 Reference Manual, Science Research Associates, Chicago.

Iverson, K. E., Elementary Functions: an algorithmic treatment, Science Research Associates, Chicago, 1966.

Falkoff, A. D., and Iverson, K. E., "The APL\360 Terminal System", in Interactive Systems for Experimental Applied Mathematics, Klerer and Reinfelds, eds., Academic Press (to appear).

In order to state that the calculation is to be performed for a specific triangle (such as the one sketched on the right) one would write the following:

```
A←3
B←4
C←5
P←A+B+C
S←P÷2
(S×(S-A)×(S-B)×(S-C))*.5
```




The actual form to be entered on the computer follows (the last line is the response):

```
      A←3
      B←4
      C←5
      P←A+B+C
      S←P÷2
      (S×(S-A)×(S-B)×(S-C))*.5
6
```

Such a sequence of calculations is called a <u>program</u>.

Three departures from conventional notation are to be noted:

1. The symbol ← replaces the equal sign in cases where it has the sense of "let $X=3$". This avoids ambiguity arising from other common uses of the equal sign (as in the statement of identities such as

$$(X+1)^2=X^2+2X+1 \ .$$

The expression $S\leftarrow P\div 2$ may be read as "$S$ is specified by the value of $P\div 2$".

2. The familiar omission of the multiplication sign is not permitted. This avoids the common confusion in $X(X+2)$, which means $X$ times $X+2$ and $F(X+2)$, which frequently means not $F$ times $X+2$ but the value of some function $F$ applied to the argument $X+2$. It also permits the use of multi-character variable names - thus $AREA$ can be used as a variable without danger of confusion with the product of the variables $A$, $R$, $E$ and $A$.

3. The <u>power</u> function is denoted by the symbol *, whereas in conventional notation this function has no symbol but is denoted by the raised position of the second argument. This change is made for convenience in typing and for uniformity - <u>every</u> operator is assigned a symbol, and that symbol may never be elided.

Basic functions. It is convenient to have, in addition to the arithmetic functions +, ×, and * already introduced, a number of other simple basic functions such as divide, subtract, maximum, etc. Table 1 shows a number of these basic functions and the symbols employed for them.

There are two points to note about the functions of Table 1:

1. Each of the functions takes two arguments and, in the interest of uniformity, each symbol appears between its arguments in any expression, just as do the familiar arithmetic symbols, e.g., $X+Y$, $X\times Y$, $X\lceil Y$, $X\leq Y$.

2. Each of the relational symbols is used as a proposition, rather than as an assertion. The difference is that a proposition may have one of two values, true or false. As is common in applied logic, the value true is represented by the number 1 and false by the number 0. Hence the expression $3\leq 7$ has a value (in this case 1), just as $3+7$ has a value (10).

The following examples show discourse employing some of the functions of Table 1:

```
      X←3
      Y←5
      X⌈Y
5
      X≥Y
0
      (X×Y)≤((X⌈Y)*2)
1
      ((X+1)*2)=((X*2)+(2×X)+1)
1
```

Arrays. The expression

```
      X←2 3 5 7 11
```

assigns to $X$ the set of five values indicated, and $X$ is called a vector of dimension 5. A vector can be indexed to select any of its elements. For example:

```
      X[3]
5
      X[4]-X[2]
4
      X[1 2 3]
2 3 5
```

Vectors have many uses in elementary mathematics and are a particularly important aid to exposition. Moreover, when presented simply as a convenient way of treating a family of variables ($X[1]$,$X[2]$, etc.) and not burdened with

| Sign | Name | Definition or example |
|---|---|---|
| + | Add | 2+3.2 ↔ 5.2 |
| - | Subtract | 2-3.2 ↔ ¯1.2 |
| × | Multiply | 2×3.2 ↔ 6.4 |
| ÷ | Divide | 2÷3.2 ↔ 0.0625 |
| ⌈ | Maximum | 3⌈7 ↔ 7 |
| ⌊ | Minimum | 3⌊7 ↔ 3 |
| * | Power | 2*3 ↔ 8 |
| ⍟ | Logarithm | 10⍟20 ↔ 1.3010299... |
| \| | Residue | $A\|B \leftrightarrow B-(|A)\times\lfloor B\div|A$ if $A\neq 0$, $0|N \leftrightarrow N$ for $N\geq 0$<br>3\|8 ↔ 2  3\|¯8 ↔ 1 |
| < | Less | Relations<br>Result is 1 if the relation holds, 0 if it does not:<br>3≤7 ↔ 1<br>7≤3 ↔ 0<br>5≠7 ↔ 1 |
| ≤ | Not greater | |
| = | Equal | |
| ≥ | Not less | |
| > | Greater | |
| ≠ | Not equal | |
| ∧ | And | |
| ∨ | Or | |
| ⍲ | Nand | |
| ⍱ | Nor | |

| $A$ | $B$ | $A\wedge B$ | $A\vee B$ | $A\barwedge B$ | $A\barvee B$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 |

TABLE 1

notions like "direction" or "direction and magnitude" which arise in one of their many applications, vectors cause students no difficulty whatsoever.

Each of the basic operations is extended to vectors element-by-element. For example:

```
      X←2 3 5 7 11
      Y←2 0 2 0 1
      X+Y
4 3 7 7 12
      X×Y
4 0 10 0 11
      X*Y
4 1 25 1 11
      2×X
4 6 10 14 22
      2*X
4 8 32 128 2048
      X*2
4 9 25 49 121
      4⌈X
4 4 5 7 11
      X>Y
0 1 1 1 1
```

Moreover, +/*X* denotes the sum of all components of *X*, and ×/*X* denotes the product of all components, and so on. They are called sum-reduction of *X* and product reduction of *X*, respectively. For example :

```
      +/X
28
      ×/X
2310
      +/(X×Y)
25
      +/(X>Y)
4
```

The use of vectors can be illustrated by reconsidering the calculation of the area of the triangle with sides of lengths 3, 4, and 5, using a single vector *L* to represent the three lengths:

```
      L←3 4 5
      P←+/L
      S←P÷2
      (S××/S-L)*.5
6
```

Note that the expression *S-L* is equivalent to 6- 3 4 5 and hence yields the vector 3 2 1. Consequently, ×/*S-L* has the value 6.

Function definition. The program

```
      P←+/L
      S←P÷2
      (S××/S-L)*.5
```

used above to calculate the area of any triangle with sides of length *L* determines a function in the sense that for any legitimate set of values assigned to the three-element vector *L*, the calculation produces a corresponding value of the area of the triangle. It is important to be able to define such a function, giving it a name (say *AREA*) and thereafter being able to use it as conveniently as the basic functions +, -, ×, *, etc. Such a definition is made as follows:

```
      ∇R←AREA L
[1]   P←+/L
[2]   S←P÷2
[3]   R←(S××/S-L)*.5
[4]   ∇
```

The function *AREA* can now be used in the ways expected of any function. For example:

```
      Q←AREA 3 4 5
      Q
6
      AREA 3 4 5
6
      144×AREA 3 4 5
864
      V←6 8 10
      AREA V
24
      AREA 5 7 12
0
      AREA 1 1 1
0.4330127019
      AREA 5 12 13
30
```

The method of definition should be clear from a study of the foregoing example: the first line is a header that shows which variable in the program following is the result (the one to the left of the arrow), which variable is the argument (that is, *L*) and what the name of the function is (*AREA*). The first ∇ (pronounced del) indicates that what follows is a definition, and the final ∇ marks the end of the definition.

The variables occurring in a definition are dummies in the following sense. In the execution of the statement

```
Q←3 AREA 3 4 5
```

the vector 3 4 5 is substituted for the argument (that is, *L*), the defining program is executed, and the value of the result *R* is assigned to *Q*. The argument of the function *AREA* can, of course, be any expression. For example,

```
     T←1 2 3
     AREA 2 + T
6
```

Trace. In order to gain a clear understanding of any defined function, it is helpful to be able to see the results produced by each line of the defining program as it is executed. This facility is provided by the *TRACE*. The trace control vector for a function *AREA* is denoted by *T∆AREA*, and the values assigned to this vector determine which lines of the program are traced when the function is executed. For example:

```
     T∆AREA←1 2 3
     Q←AREA 1 1 1
AREA[1] 3
AREA[2] 1.5
AREA[3] 0.4330127019
```

If the trace control is set to the value 2, then only line 2 is traced:

```
     T∆AREA←2
     Q←AREA 1 1 1
AREA[2] 1.5
```

Iteration. Repetition of some operation (called iteration) is an important notion in mathematics. For example, the power function *X***N* is defined as an iteration of multiplication extended to *N* factors each having the value *X*.

Iteration is used in the following function definition:

```
      ∇Z←B N
[1]   Z←1
[2]   D←0,Z
[3]   E←Z,0
[4]   Z←D+E
[5]   →2
[6]   ∇
```

Two points merit comment:

1. The notation $\rightarrow 2$ occuring on line 5 is read as "branch to 2" and causes line 2 to be executed next in sequence. Hence the sequence of lines 2, 3, 4, and 5 will be executed repeatedly.

2. The comma denotes <u>catenation</u>. For example:

```
      X←1 2 3
      Y←4 1
      X,Y
1 2 3 4 1
      Y,X
4 1 1 2 3
      0,X
0 1 2 3
      0,1
0 1
```

The behavior of the function $B$ should be apparent from the following trace:

```
     TΔB←1 2 3 4 5
      Q←B 3
B[1] 1
B[2] 0 1
B[3] 1 0
B[4] 1 1
B[5] 2
B[2] 0 1 1
B[3] 1 1 0
B[4] 1 2 1
B[5] 2
B[2] 0 1 2 1
B[3] 1 2 1 0
B[4] 1 3 3 1
B[5] 2
B[2] 0 1 3 3 1
B[3] 1 3 3 1 0
B[4] 1 4 6 4 1
B[5] 2
B[2] 0 1 4 6 4 1
B[3] 1 4 6 4 1 0
B[4] 1 5 10 10 5 1
B[5] 2
B[2] 0 1 5 10 10 5 1
B[3] 1 5 10 10 5 1 0
B[4] 1 6 15 20 15 6 1
```

It is clear that the value of $Z$ produced (on line 4) at the Kth iteration is the vector of binomial coefficients of order K.

Because the branch to line 2 occurring on line 5 of the function *B* is unconditional, the execution of the program would continue indefinitely unless stopped (as it was in the foregoing example) by striking the attention key located at the upper right corner of the keyboard. A conditional branch can be made by replacing the 2 occurring in line 5 by a suitable expression.

The definition of the function *B* can be changed by re-opening the definition and introducing the new values of any revised lines as follows:

```
      ∇B
[6]   [5]
[5]   →2×N>Z[2]
[6]
```

The entire revised function may now be displayed:

```
      [□]∇
    ∇ Z←B N
[1]   Z←1
[2]   D←0,Z
[3]   E←Z,0
[4]   Z←D+E
[5]   →2×N>Z[2]
    ∇
```

Execution of the revised function (with trace) will now illustrate the behavior of the conditional branch:

```
      T∆B←1 2 3 4 5
      Q←B 4
B[1] 1
B[2] 0 1
B[3] 1 0
B[4] 1 1
B[5] 2
B[2] 0 1 1
B[3] 1 1 0
B[4] 1 2 1
B[5] 2
B[2] 0 1 2 1
B[3] 1 2 1 0
B[4] 1 3 3 1
B[5] 2
B[2] 0 1 3 3 1
B[3] 1 3 3 1 0
B[4] 1 4 6 4 1
B[5] 0
```

Since the result of the function was assigned to $Q$ (that is, $Q\leftarrow B\ 4$), the value of $Q$ should be the last value assigned to the result $Z$:

```
      Q
1 4 6 4 1
```

It is clear from the trace of line 5 that the last branch is made to line 0 rather than to 2. Since the function has no line 0, this causes termination of the execution.

If the trace control is set to discontinue tracing, the function $B$ can then be used to produce binomial coefficients without producing intermediate output:

```
      T∆B←0
      Q←B 4
      Q
1 4 6 4 1
      B 4
1 4 6 4 1
      B 6
1 6 15 20 15 6 1
      B 10
1 10 45 120 210 252 210 120 45 10 1
```

Order of execution. In the expression $(X+Y)\times(X-Y)$, the parentheses determine the order in which the functions are evaluated. In the present notation, parentheses are used for this purpose exactly as they are in familiar algebraic notation.

In algebraic notation, there is a rather complex and ill-defined set of rules which determines the order of evaluation in the absence of parentheses. In the present notation there is one simple rule - every function takes as its right argument the entire expression to the right of it. For example,

```
      3×X⌈Y+P*Q≤5
```

is equivalent to

```
      3×(X⌈(Y+(P*(Q≤5))))
```

This simple rule has four happy consequences:

1. An expression is easy to read from left to right - e.g., the above example is clearly 3 times something, that thing is the maximum of *X* and something, and so on.

2. This analysis from left to right is familiar in modern Romance languages. For example: "They objected to the rise in price of products from the farm" means "They objected to" something, that thing is "the rise in" something, and so on.

3. An expression is also easy to read from right to left, since an equivalent statement of the rule is that the functions are evaluated in order from right to left. (See the parenthesized form of the example above.)

4. With this rule of evaluation, the expression -/*X* yields the alternating sum of the elements of *X*, and ÷/*X* yields the alternating product. For example, if *X*←2 3 5 7 11, then -/*X* is equivalent to 2-3-5-7-11. With the present rule, this is equivalent to (2+5+11)-(3+7).

Functions of two arguments. Each of the functions defined thus far take a single argument. The definition of a function of two arguments will be illustrated by defining a function *POL* which takes a left argument *C* (a vector of coefficients) and a right argument *X*, and yields the value of the polynomial with coefficients *C* evaluated at the point *X*:

```
     ∇Z←C POL X
[1]  Z←+/C×X*¯1+⍳⍴C
[2]  ∇

     1 2 3 POL 2
17
     1 2 3 POL 3
34
     1 3 3 1 POL 4
125
```

Two symbols used in the definition require explanation. The function ⍴*Q* yields the dimension of its argument, that is, ⍴*Q* is the number of elements in the vector *Q*. For example:

```
     C←1 3 3 1
     ⍴C
4
```

The function ⍳ applied to the argument $N$ yields the vector of the first $N$ integers in order:

```
      ⍳3
1 2 3
      ⍳4
1 2 3 4
      ⍳⍴C
1 2 3 4
      ¯1+⍳⍴C
0 1 2 3
      X←4
      X*¯1+⍳⍴C
1 4 16 64
      C
1 3 3 1
      C×X*¯1+⍳⍴C
1 12 48 64
      +/C×X*¯1+⍳⍴C
125
```

The foregoing examples show that the expression $\bar{}1+\iota\rho C$ yields the vector of exponents appropriate to the coefficients $C$, and $X*\bar{}1+\iota\rho C$ is the vector of powers of $X$. Hence $C\times X*\bar{}1+\iota\rho C$ is the vector of terms of the polynomial, and $+/C\times X*\bar{}1+\iota\rho C$ is their sum.

Conclusion. This concludes the introduction of the main features of the notation to be employed herein: the basic functions and the symbols used to denote them (Table 1); the use of vectors, including the element-by-element extension of the basic functions to vectors and the reduction of a vector by applying some function like + to all elements; the definition and naming of new functions; tracing the execution of a function; and the order of execution in unparenthesized expressions. A few further details of the notation will be introduced as needed in examples.

## EXERCISES AND EXPERIMENTATION

In the physical sciences, experiments have long been accepted as an essential part of the educational process. Closely-directed experiments can serve to develop intuition, that is, to furnish the student with concrete models of abstract notions. Freer experimentation can be used to confront the student with an unfamiliar system and a challenge to learn its secrets through his own choice of experiments. Because of the human penchant for puzzles, the opportunity for this type of experimentation can provide a strong incentive to study, developing both the student's taste for, and techniques of, exploration.

Experimentation can play similar roles in other disciplines as well. In mathematics, for example, experiments in plotting quadratics with various coefficients can give a student a feeling for the behavior of parabolas. Other suggested experiments can lead students to the discovery of properties of mathematical objects; for example, the calculation of the direction of rays reflected from a parabolic mirror leads naturally to the interesting properties of the focal point.

Such mathematical experiments have typically been performed by pencil-and-paper calculations, a method too tedious and time-consuming to admit the assignment of much non-trivial experimentation. The computer, however, now makes large-scale mathematical experimentation feasible and for the first time provides the student with a usable "mathematical laboratory".

Even in the physical sciences, the computer can, through the use of mathematical models, provide an important supplement to experiments with physical equipment. The ease and rapidity of experiments on mathematical models can quickly give a student an intuitive feeling for the behavior of a system which could only be attained by long and tedious work with the corresponding physical experiments.

Exercises can be used simply to enhance facility in some process already well-understood, as in drill exercises in multiplication. However, when used to elicit or elucidate new ideas, assigned exercises are essentially guided experimentation - a well-designed set of exercises guides the student through a sequence of experiments.

The remainder of this section will be devoted to a set of exercises chosen to illustrate various facets of the use of the computer in experimentation.

Simple experiments. For beginning students, a good deal of motivation can be provided by simply leaving most of the basic functions as puzzles, the definition of each function to be determined by experimentation. For example, the student is first shown the sequence

```
      3.4+5.6
9
      3.4×6
20.4
```

and is then told that the symbols ÷-*⌈⌊<≤=≥>≠ are also functions of two arguments to be used like the + and ×. He is then invited to determine what these functions are by trying them on the computer with arguments of his own choice.

This can prove an interesting and (for younger students) challenging exercise. Observations of students in this simple exercise have led me to two conclusions:

1. There is commonly an initial reluctance to experiment. Rather than plunge into trials, the student asks "What will happen if I ...", or "Is it all right if I ...". This suggests that our treatment of students frequently succeeds in stifling any urge to explore matters on their own.

2. Students lack techniques of systematic exploration. For example, one student who quickly identified *X***Y* as the *Y*th power of *X*, spent a much longer time on the essentially simpler function ⌈ (maximum) and then wrongly concluded that the value of *X*⌈*Y* was simply the value of the right argument *Y*. Although he had tried quite a number of experiments, all had been of the form *X*⌈*Y* where *Y* was the larger. I would have chosen experiments in pairs of the form

```
      3⌈8
8
      8⌈3
8
```

and was surprised that he had not, until I realized that my choice of such a sequence was based on an appreciation of the importance of commutativity, an appreciation that he had not gained in spite of exposure to it for a year or more in classes.

The following exercise can be used to help fix the concepts of associativity, commutativity, and distributivity: perform experiments to determine which of the functions +, ×, ⌈, ⌊, and * are commutative, which are associative, and which of the functions distribute over which functions. The solution to the exercise is shown in Table 2.

Subsequent examples employ the function ⍳, whose behavior is recalled here:

```
      ⍳3
1 2 3
      .2×⍳10
0.2 0.4 0.6 0.8 1 1.2 1.4 1.6 1.8 2
```

Interest in the generalizations of familiar functions (such as the generalization of *X***N* to non-integer exponents *N*) can be sparked by first assigning suggestive experiments.

**Commutativity**

| + | - | × | ÷ | ⌈ | ⌊ | * |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 1 | 0 |

**Associativity**

| + | - | × | ÷ | ⌈ | ⌊ | * |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 1 | 0 |

**Distributivity**

| | + | - | × | ÷ | ⌈ | ⌊ | * |
|---|---|---|---|---|---|---|---|
| + | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| - | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| × | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| ÷ | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ⌈ | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| ⌊ | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| * | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Fundamental properties of some functions**

TABLE 2

For example, the student who is already familiar with the following behavior of the power function:

```
      X←⍳4
      X
1 2 3 4
      2*X
2 4 8 16
```

can then be urged to plot the results of experiments of the following form:

```
      X←.5×⍳6
      X
0.5 1 1.5 2 2.5 3
      2*X
1.414213562 2 2.828427125 4 5.656854249 8
      2*X÷2
1.189207115 1.414213562 1.681792831 2 2.37841423 2.82842712
3*X
      1.732050808 3 5.196152423 9 15.58845727 27
```

Similar experiments can be used in the extension of the factorial function (denoted by *!N*):

```
      !3
6
      !5
120
      !1 2 3 4
1 2 6 24 120
      !⍳5
1 2 6 24 120
      !.5×⍳10
0.8862269254 1 1.329340388 2 3.32335097 6 11.6317284 24
```

Investigation of new functions. A teacher can define any set of new functions desired (in the manner illustrated by the functions *AREA* and *B* defined earlier) and can then save the entire set under some name (say *FRODO*) by typing the first line below:

```
     )SAVE FRODO
FRODO SAVED 03/12/68  30.46.33
```

Thereafter, anyone (student or teacher) who types the first line below:

```
     )LOAD FRODO
FRODO SAVED  03/12/68  30.46.33
```

will again have available the set of defined functions.

Suppose a teacher defines and saves the following function:

```
     ∇Z←BIN N
[1]  Z←1
[2]  Z←(0,Z)+Z,0
[3]  →2×N≥ρZ
[4]  ∇
```

A student may then be asked to load the function and, without displaying its definition, determine what the function is. A few experiments like the following:

```
     BIN 1
1 1
     BIN 2
1 2 1
     BIN 5
1 5 10 10 5 1
```

should suffice to identify *BIN* as a function which produces the binomial coefficients of order *K* when applied to the argument *K*.

The student may now be asked to analyze the definition of the function *BIN* by displaying it. As an aid to the analysis he might execute it with a trace applied to line 2:

```
     T∆BIN←2
     Q←BIN 5
BIN[2] 1 1
BIN[2] 1 2 1
BIN[2] 1 3 3 1
BIN[2] 1 4 6 4 1
BIN[2] 1 5 10 10 5 1
```

From this the student should see the relation to Pascal's triangle. He could, however, get an even more detailed view by breaking line 2 into an equivalent sequence of three statements:

```
     D←0,Z
     E←Z,0
     Z←D+E
```

This change in the definition of *BIN* can be made as follows:

```
      ∇BIN
[4]   [1.1]D←0,Z
[1.2] E←Z,0
[1.3] [2]Z←D+E
[3]   ∇
```

A subsequent display of *BIN* would show its new definition:

```
      ∇BIN[□]∇
    ∇ Z←BIN N
[1]   Z←1
[2]   D←0,Z
[3]   E←Z,0
[4]   Z←D+E
[5]   →2×N≥ρZ
    ∇
```

This is clearly equivalent to the definition of the function *B* already used in the introduction, and a trace of its execution (also shown in the introduction) produces the desired detailed view.

A function definition is usually closed with the symbol ∇. If the symbol ⍫ is used instead, the function becomes locked in the following sense: (1) its definition cannot be reopened and hence it can be neither modified or displayed, and (2) a trace cannot be applied to it. By locking its definition, the teacher can therefore present a student with a function whose behavior can be studied only by the means available for the basic functions (⌈, ⌊, *, etc.), that is, by experimentation.

Theory. The theory involved in elementary mathematics is largely concerned with establishing identities, that is, with proving that two different computational procedures applied to the same argument yield the same result for all possible values of the argument. Experiments on functions can lead the student to the discovery of such identities. Moreover, the structure of the program defining a function frequently suggests general proofs of observed properties of the function.

For example, experiments of the form

```
      X←2
      N←4
      (X+1)*N
81
       (BIN N)POL X
81
```

repeated for various integral values of *N* and *X* would suggest that the polynomial in *X* whose coefficients are the binomial coefficients of order *N* is equivalent to *X*+1 raised to the *N*th power, that is,

$$(X+1)*N \leftrightarrow (BIN\ N)POL\ X \qquad \text{(Eq. 1)}$$

Similarly, experiments of the form

```
      +/BIN 1
2
      +/BIN 2
4
      +/BIN 3
8
      +/BIN 4
16
      +/BIN 15
32768
      2*15
32768
```

suggest the identity:

```
      (+/BIN N) ↔ 2*N
```

The usual proof is based on considering Equation 1 for the case $X=1$. A more direct proof is provided by the original definition of the function *BIN*, namely:

```
      ∇Z←BIN N
[1]   Z←1
[2]   Z←(0,Z)+Z,0
[3]   →2×N≥ρZ
[4]   ∇
```

It is clear that the effect of step 3 is to repeat the second step $N$ times. Furthermore, it is clear that each execution of step 2 doubles the value of $+/Z$ (the sum of the coefficients), since the new value of $Z$ is formed by adding $Z$ to itself and to two zeros.

Consider the function *G* defined and used as follows:

```
      ∇R←M G N
[1]   R←M
[2]   M←M|N
[3]   N←R
[4]   →M≠0
[5]   ∇

      40 G 48
8
      120 G 84
12
      84 G 120
12
```

Further experiments can lead to the (correct) conjecture that *G* yields the greatest common divisor of its arguments. Its detailed execution can be seen in the following trace:

```
      TΔG←1 2 3 4
      Q←84 G 120
G[1] 84
G[2] 36
G[3] 84
G[4] 1
G[1] 36
G[2] 12
G[3] 36
G[4] 1
G[1] 12
G[2] 0
G[3] 12
G[4] 0

      Q
12
```

A study of this trace shows that the effect of each iteration is to replace the larger argument by the remainder obtained on dividing the larger by the smaller - in other words, the function definition is a concise statement of the well-known Euclidean algorithm. Experiments with *G* therefore suggest the theorem underlying the Euclidean algorithm, namely that the greatest common divisor of two arguments is the same as the greatest common divisor of one of them and their remainder.

Exercises in composing functions. The composition of formal function definitions is called programming. In the exercises treated thus far the student was only required to use and study functions which had already been programmed for him. It is also important that the student learn to program.

Functions to be formally defined can either be presented informally (e.g., "define a function to yield the vector of the first *N* primes") or formally by presenting a function whose behavior is to be emulated.

In this work the computer can be used to experiment with the function being developed so as to identify and correct deficiencies. The main lesson to be learned is precision in thought and expression.

The following experience with a high school senior will illustrate the process of programming. The following problem was posed: define a function to determine the reduced form of a rational fraction, i.e., given two

integers $A$ and $B$, determine integers $M$ and $N$ such that $M \div N$ equals $A \div B$ and that $M$ and $N$ have no common factor. When asked to state the method he used for such problems, the student said he would factor both integers, strike out the common factors, and then take the products of the remaining factors of each.

However, when given the pair 28 and 70, he immediately answered 2 and 5, and when asked for his method replied "I saw that 7 was a factor of each so I divided it out and then recognized that 2 was a factor of the results and then divided it out" - the actual method used was not the one professed. When pressed on this discrepancy, the student decided the method he actually used was the better one. When presented with more difficult cases he soon developed a systematic procedure, trying to divide first by 2, then by 3, and so on.

The student was then asked to state the process in formal terms and proceeded (with the aid of occasional suggestions) to reason roughly as follows: Call the trial divisor $T$. It must first be set to some initial value. To start a new trial, $T$ will have to be increased, the remainder on dividing it into $A$ will then be compared with zero to see if $T$ is a factor of $A$. If it is not, then a new trial must be begun by repeating the step of incrementing $T$. Thus:

```
      ∇F
[1]   T←1
[2]   T←T+1
[3]   →2×⍳0≠T|A
[4]
```

(The effect of line 3 is to branch to 2 if $T$ is not a factor of $A$, and to "fall through" to line 4 otherwise. This occurs because ⍳1 yields the value 1, whereas ⍳0 is an empty vector and no branch occurs in that case.)

Following the test of $A$ on line 3, a similar test on $B$ is required. If both tests show divisibility, then both $A$ and $B$ are respecified by dividing through by $T$, and the process is repeated on the new values of $A$ and $B$. Thus:

```
      ∇F
[1]   T←1
[2]   T←T+1
[3]   →2×⍳0≠T|A
[4]   →2×⍳0≠T|B
[5]   A←A÷T
[6]   B←B÷T
[7]   →3
[8]   ∇
```

The following experiment - with complete trace - was then performed:

```
      TΔF←⍳7
      A←84
      B←360
      F
F[1] 1
F[2] 2
F[3]
F[4]
F[5] 42
F[6] 180
F[7] 3
F[3]
F[4]
F[5] 21
F[6] 90
F[7] 3
F[3] 2
F[2] 3
F[3]
F[4]
F[5] 7
F[6] 30
F[7] 3
F[3] 2
F[2] 4
F[3] 2
F[2] 5
F[3] 2
F[2] 6
F[3] 2
F[2] 7
F[3]
F[4] 2
F[2] 8
```

(The attention key was used to interrupt execution as soon as it became apparent that the process would never terminate. The student recognized the need for a termination test and decided to insert a comparison of *T* and the minimum of *A* and *B* immediately after the incrementation of *T* on step 2. The final program appears below.)

```
      ∇ F
[1]    T←1
[2]    T←T+1
[3]    →0×⍳T>A⌊B
[4]    →2×⍳0≠T|A
[5]    →2×⍳0≠T|B
[6]    A←A÷T
[7]    B←B÷T
[8]    →3
     ∇

       T∆F←0
       A←84
       B←360
       F
       A
7
       B
30
```

From the outset the student recognized the desirability of using only successive primes rather than successive integers for the successive values of $T$, but this refinement was suppressed in the interest of simplicity. He did, however, initially make the mistake of branching to line 2 rather than 3 from line 8. The effect was to miss repeated factors (e.g., dividing out a 3 must be followed by a test for further factors of 3 before proceeding to the next trial divisor). This mistake was quickly caught by experimentation and then rectified.

Since the arguments $A$ and $B$ receive identical treatment in the process, it is clear that they might conveniently be treated as the two elements of a single vector $P$. When this was pointed out to the student, he designed the following process, which parallels the original in every particular:

```
      ∇F2
[1]   T←1
[2]   T←T+1
[3]   →0×⍳T>⌊/P
[4]   →2×⍳⌈/0≠T|P
[5]   P←P÷T
[6]   →3
[7]   ∇

      P←84 360
      F2
      P
7  30
```

Flowcharting. In planning any formal procedure (i.e., defining a function), it is rather common practice to construct a flowchart, consisting of informal statements of the parts of the procedure together with arrows showing the sequence in which the parts are to be executed. Figure 1, for example, shows a flowchart which describes the following process: determine the maximum value of each pair of corresponding elements of the vectors $X$ and $Y$, and then determine the minimum of the resulting set of maxima.

The main disadvantage of the flowchart is that it is an informal (or at best a poorly-specified formal) language, and its use does not provide the discipline and precision of a formal language. It also tends to be more diffuse and less perspicuous - compare , for example, the flowchart of Figure 1 with the equivalent *APL* statement, $\lfloor/X\lceil Y$.

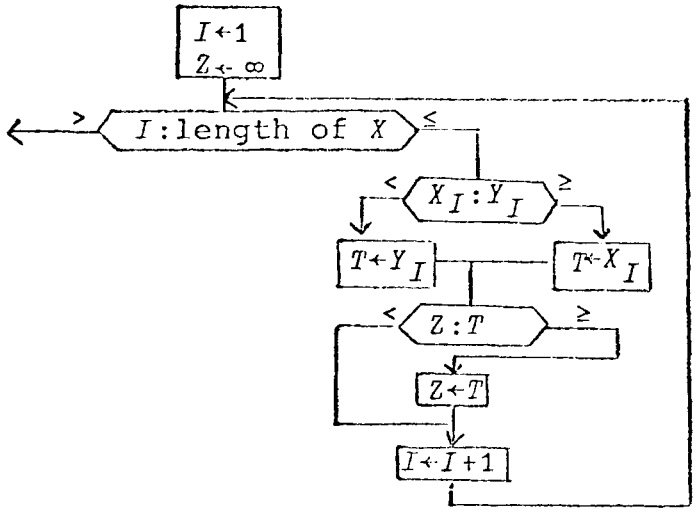
The flowchart does embody two useful notions:

1. The use of arrows to give a graphic picture of the sequence of execution in a program.

2. The ability to name and use a process in the overall planning of a procedure before the process itself is defined in detail.

However, both of these notions can be employed within the confines of a formal language. The informal use of branch arrows in addition to the formal expression of branches is helpful in any formal language and should be encouraged. The ability to name and use functions before defining them is already inherent in any formal language which incorporates methods for defining new functions, and the use of this ability is also to be encouraged.

In sum, flowcharting shows no advantage over a well-designed formal language.




Flowchart of $\lfloor/X\lceil Y$

Figure 1

Non-mathematical work. Non-mathematical work amenable to the application of a computer covers a wide range of areas including sorting, plotting, text analysis, and the description of the internal operation of the computer itself. Only the first three will be illustrated here.

The following bits of new notation will be employed in subsequent examples:

1. Any string of characters enclosed in quotes denotes a vector whose successive elements are the successive characters in the string:

```
      W←'CAT'
      W[2]
A
      W[2 1 3]
ACT
```

2. If $U$ is a logical vector (comprising elements of zeros and ones only), then $U/X$ denotes a selection of those elements of $X$ corresponding to the ones in $U$. The operation is called compression:

```
      1 0 1 0 1/2 3 5 7 11
2 5 11
      1 0 1 0 1/'ABCDE'
ACE
      X←5 3 9 3 14 6
      (X=⌊/X)/X
3 3
       (X≠⌊/X)/X
5 9 14 6
```

3. If $B$ is a vector, then $B⍳X$ denotes the index of $X$ in $B$:

```
      B←2 3 5 7 11
      B⍳5
3
      B⍳5 2 7
3 1 4
      A←'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
      A⍳'CAT'
3 1 20
      A[3 1 20]
CAT
```

(It should be noted that the symbol ⍳ has already been used for a function of one argument (e.g., ⍳3 is 1 2 3) and is here used as a function of two arguments. This is similar to the familiar double usage of the minus sign to denote both subtraction ($X-Y$) and negation ($-Y$) and does not introduce an ambiguity.)

The need to sort a list of numbers into ascending order arises frequently in a variety of areas (e.g., sorting account numbers into order for posting to ledgers). There exists a large variety of methods for sorting. One of the simplest (but inefficient) methods may be stated as follows: locate the smallest item remaining to be sorted, append it to the list of sorted items and remove it from the list of items remaining to be sorted. A formal statement follows:

```
    ∇ Z←S X
[1]   Z←⍳0
[2]   Z←Z,(X=⌊/X)/X
[3]   X←(X≠⌊/X)/X
[4]   →2×0≠⍴X
    ∇
```

The following complete trace should clarify the process:

```
      T∆S←⍳4
     S 5 3 2 16 3 8
S[1]
S[2] 2
S[3] 5 3 16 3 8
S[4] 2
S[2] 2 3 3
S[3] 5 16 8
S[4] 2
S[2] 2 3 3 5
S[3] 16 8
S[4] 2
S[2] 2 3 3 5 8
S[3] 16
S[4] 2
S[2] 2 3 3 5 8 16
S[3]
S[4] 0
2 3 3 5 8 16
```

The problem of sorting non-numeric data will be illustrated by sorting the following text vector:

```
    T←'OLAF ( UPON WHAT ONCE WERE KNEES) DOES ALMOST
CEASELESSLY REPEAT'
```

It is first necessary to specify the order of the alphabet assumed (although the ordering of the letters is well established, the ordering of the space, hyphen, punctuation marks, and special symbols is not):

```
    A←'ABCDEFGHIJKLMNOPQRSTUVWXYZ- ( ) *,.;: !
```

The operation $A\iota T$ can now be used to determine for each element of $T$ its position in the alphabet:

```
      J←AιT
      J
15  12  1  6  28  29  21  16  15  14  28  23  8  1  20
      28  15  14  3  5  28  23  5  18  5  28  11  14
      5  5  19  30  28  4  15  5  19  28  1  12  13
      15  19  20  28  3  5  1  19  5  12  5  19  19
      12  25  28  18  5  16  5  1  20
```

The function $S$ can now be applied to sort the numeric vector $J$:

```
      K←S J
      J
1  1  1  1  1  3  3  4  5  5  5  5  5  5  5  5  5  5
      5  6  8  11  12  12  12  12  13  14  14  14  15
      15  15  15  15  16  16  18  18  19  19  19  19
      19  19  20  20  20  21  23  23  25  28  28  28
      28  28  28  28  28  28  29  30
```

The final sorted output (in terms of the original alphabet) can now be obtained as follows:

```
      Q←A[K]
      Q
AAAAACCDEEEEEEEEEEEEFHKLLLLMNNNOOOOOPPRRSSSSSSTTTUWWY
```

The entire process can be seen more clearly in the following single statement:

```
      Q←A[S AιT]
```

A sorted list of the letters occurring in the text $T$ (without repetition) can be obtained by modifying the sorting program $S$ so that only one occurence of the minimum is appended to the result at each excution of line 2:

```
      ∇S[2□]
[2]     Z←Z,(X=⌊/X)/X
[2]     Z←Z,⌊/X
[3]     [□]∇
    ∇ Z←S X
[1]     Z←ι0
[2]     Z←Z,⌊/X
[3]     X←(X≠⌊/X)/X
[4]     →2×0≠ρX
    ∇
      S AιT
1  3  4  5  6  8  11  12  13  14  15  16  18  19  20
      21  23  25  28  29  30
      A[S AιT]
ACDEFHKLMNOPRSTUWY ()
```

The problem of plotting will be illustrated by producing a histogram of the vector *V*, that is, the height of the ordinate at the *K*th point on the abscissa is to be *V*[*K*]. For example, if

```
V←1 2 3 5 8 13 11 9 7 5 5 8 10 12 8 4 2 1 1 1
```

then the lines of the plot for the ordinate values of 7, 6, and 5, can be obtained as

```
      7≤V
0 0 0 0 1 1 1 1 1 0 0 1 1 1 1 0 0 0 0 0
      6≤V
0 0 0 0 1 1 1 1 1 0 0 1 1 1 1 0 0 0 0 0
      5≤V
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0
```

where the ones denote points to be plotted, and the zeros denote spaces. The following is a formal statement of the process:

```
      ∇PLOT X
[1]   I←⌈/X
[2]   I≤X
[3]   I←I-1
[4]   →2×0<I
[5]   ∇
```

For example:

```
      PLOT V
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 1 1 0 0 0 0 0 1 1 0 0 0 0 0 0
0 0 0 0 0 1 1 1 0 0 0 0 1 1 0 0 0 0 0 0
0 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 0 0
0 0 0 0 1 1 1 1 1 0 0 1 1 1 1 0 0 0 0 0
0 0 0 0 1 1 1 1 1 0 0 1 1 1 1 0 0 0 0 0
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

A neater plot can be obtained by changing line 2 of the program as follows:

```
      ∇PLOT[2] ' ⊥'[1+I≤X]∇
```

The effect is to substitute a space for each 0 and a base symbol for each 1:

```
      ∇PLOT[⎕]∇
     ∇ PLOT X
[1]    I←⌈/X
[2]    ' ⊥'[1+I≤K]
[3]    I←I-1
[4]    →2×0<I

      PLOT V
      ⊥
      ⊥        ⊥
      ⊥⊥       ⊥
      ⊥⊥      ⊥⊥
      ⊥⊥⊥     ⊥⊥
     ⊥⊥⊥⊥    ⊥⊥⊥⊥
     ⊥⊥⊥⊥⊥   ⊥⊥⊥⊥
     ⊥⊥⊥⊥⊥   ⊥⊥⊥⊥
    ⊥⊥⊥⊥⊥⊥⊥⊥⊥⊥⊥⊥
    ⊥⊥⊥⊥⊥⊥⊥⊥⊥⊥⊥⊥⊥
   ⊥⊥⊥⊥⊥⊥⊥⊥⊥⊥⊥⊥⊥⊥
  ⊥⊥⊥⊥⊥⊥⊥⊥⊥⊥⊥⊥⊥⊥⊥
⊥⊥⊥⊥⊥⊥⊥⊥⊥⊥⊥⊥⊥⊥⊥⊥⊥
```

The problems of text analysis will be illustrated by counting the number of occurrences of each letter in the text vector *T* used in the discussion of sorting:

```
       T
OLAF ( UPON WHAT ONCE WERE KNEES) DOES ALMOST
       CEASELESSLY REPEAT
       T='E'
0  0   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
       0  1  0  0  1  0  1  0  0  0  1  1  0  0  0  0
       0  1  0  0  0  0  0  0  0  0  0  0  1  0  0  1
       0  1  0  0  0  0  0  0  1  0  1  0  0

    +/T='E'
11
```

The vector result above clearly has a one for each occurrence of the letter *E* in the text *T*. The final result (11) is the sum of these ones and is therefore the number of *E*'s in *T*.

A similar count for each letter (or symbol) in the alphabet *A* can clearly be obtained by writing a program to treat each element *A*[*I*] of the alphabet in turn. It can be done more conveniently with the aid of the outer product.

Observe the following example of the outer product of two vectors *X* and *Y*:

```
      X←2 3 4
      Y←3 7 5 3 2
      X∘.×Y
 6  14  10  6  4
 9  21  15  9  6
12  28  20 12  8
```

The result is a matrix, the element in the *I*th row and *J*th column has the value *X*[*I*] × *Y*[*J*]. More generally, any other operator can be substituted for the ×. For example:

```
      X∘.≤Y
1  1  1  1  1
1  1  1  1  0
0  1  1  0  0

      'ABCDEFG'∘.='CABBAGE'

0  1  0  0  1  0  0
0  0  1  1  0  0  0
1  0  0  0  0  0  0
0  0  0  0  0  0  0
0  0  0  0  0  0  1
0  0  0  0  0  0  0
0  0  0  0  0  1  0
```

The first row shows the occurrences of the letter *A*, the second shows the occurences of *B*, etc. Since the summation +/*M* applied to a matrix *M* sums the rows of *M*, the expression:

```
      +/'ABCDEFG'∘.='CABBAGE'
2  2  1  0  1  0  1
```

yields the counts of *A*,*B*,*C*, etc. in the word *CABBAGE*.

Returning now to the text *T* and alphabet *A*, the letter counts in *T* can clearly be obtained as follows:

```
      +/A∘.=T
5  0  2  1  11  1  0  1  0  0  1  4  1  3  5  2  0  2
      6  3  1  0  2  0  1  0  0  9  1  1  0  0  0  0  0
```

Heuristic functions. Certain functions prove particularly effective in "leading to discovery". Some such functions are useful only in exploring a particular phenomenon, whereas others apply to a rather wide class. Examples of both types will be treated.

The factor by which the principal amount of a loan is increased when loaned at interest $I$ compounded yearly for $Y$ years is clearly given by the expression $(1+I)*Y$. If interest is compounded $N$ times per year, the corresponding expression becomes $(1+I\div N)*N\times Y$. Exploration of the limiting value of this expression for large values of $N$ (and for $Y=1$) leads to the exponential function of the argument $I$.

Considering first the simple case for $I=1$ and $Y=1$, the expression becomes $(1+1\div N)*N$. Experiments could now be performed for various values of $N$, but tedium can be avoided by defining the following function:

```
      ∇Z←E N
[1]   Z←(1+1÷N)*N
[2]   ∇
```

Experiments can now be performed conveniently:

```
      E 1
2
      E 10
1.59374246
      E 100
2.704813829
      E 1000
2.716923932
```

Moreover, vector arguments can be used:

```
      E 1 10 100 1000
2  2.59374246  2.704813829  2.716923932
      E 10*0,⍳6
2  2.59374246  2.704813829  2.716923932  2.718145927
      2.718268237  2.718280469
```

Differencing is a generally useful method of studying a function - it consists of calculating a vector of values of the function for a set of equally spaced arguments and then determining the differences between successive values of the function. This can be done conveniently with the aid of the following function:

```
      ∇Z←DF Y
[1]   Z←Y[1+⍳¯1+⍴Y]-Y[⍳¯1+⍴Y]
[2]   ∇
```

The result of this function is clearly a vector of dimension one less than its argument $Y$ whose $I$th component has the value $Y[I+1]-Y[I]$.

Consider the application of *DF* to the vector of values *V* obtained by applying the "square" function to the vector *X* of integers 1 to 100:

```
      X←⍳10
      V←X*2
      V
1  4  9  16  25  36  49  64  81  100
      DF V
3  5  7  9  11  13  15  17  19
```

The obvious pattern in the last result suggests an easily proved theorem. The pattern for the "cubes" function is not so evident:

```
      V←X*3
      DF V
7  19  37  61  91  127  169  217  271
```

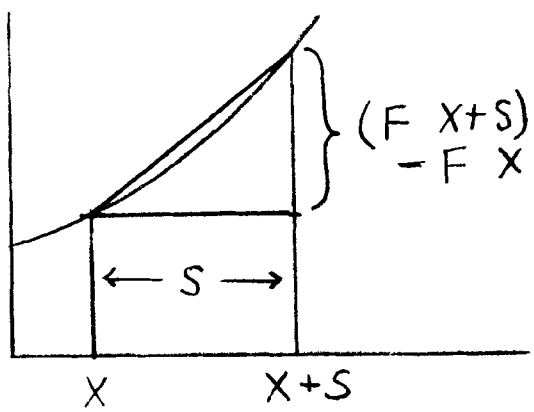
However, the second difference (obtained by applying *DF* to the first difference) shows a marked pattern:

```
      DF DF V
12  18  24  30  36  42  48  54
      DF DF DF V
6  6  6  6  6  6  6
      DF DF DF DF V
0  0  0  0  0  0
```

These results suggest theorems about the cube, and also suggest experiments on further functions such as the higher powers, the general polymonial, and the triangular numbers.

The slope of the tangent to a curve (i.e., the derivative of the function represented by the curve) is also important in the study of a function. The tangent slope at the point (*X*, *F X*) is approximated by the slope of the secant through the points (*X*, *F X*) and ((*X*+*S*), *F X*+*S*) which (as seen from the accompanying sketch) is given by the expression ((*F X*+*S*)-*F X*)÷*S*. Hence the following function yields the secant slope:

```
      ∇Z←S SL X
[1]   Z←((F X+S)-F X)÷S
[2]   ∇
```



For example, if F is defined as the square function:

```
      ∇Z←F X
[1]   Z←X*2
[2]   ∇
```

then the secant slope at points with abscissae 2 and 2+.5 is:

```
      .5 SL 2
4.5
```

Moreover:

```
       S←10*-0,⍳5
       S
1  0.1  0.01  0.001  0.0001  1E¯5
       S SL 2
5  4.1  4.01  4.001  4.0001  4.00001
```

The last vector gives the secant slopes for the successive spacings 1, 0.1, 0.01, etc., and suggests that the limiting value (i.e., the tangent slope) is 4.

Once the notion of this limiting slope is accepted, the student may experiment with a fixed small spacing (say 10*¯6) and with a vector of values of the argument X so as to see the slope at various points on the curve. Thus the experiment

```
       S←10*¯6
1E¯6
       X←⍳8
       X
1  2  3  4  5  6  7  8
       S SL X
2.000000997  4.000000999  6.000000997  8.000000946
       10.000001  12.00000099  14.00000099  16.00000098
       2×X
2  4  6  8  10  12  14  16
```

suggests (correctly) that the slope of the tangent to the function X*2 at any point P is 2×P.

To experiment with other functions it is only necessary to change the definition of F. For example:

```
      ∇F[1]Z←X*3∇
```

makes *F* the cube function. Hence:

```
      .000001 SL ⍳8
3.000002991  12.00000598  27.00000897  48.00001193
      75.00001487  108.0000179  147.0000208  192.0000238
      3×X*2
3  12  27  48  75  108  147  192
```

suggests a theorem concerning the slope of the cube function.

The polynomial function *POL* defined in the introduction can be used for general experiments with power series. Its definition will first be recalled:

```
      ∇POL[⎕]∇
     ∇ Z←C POL X
[1]    Z←+/C×X*¯1+⍳⍴C
     ∇

      1 3 3 1 POL 2
27
```

(The left argument determines the vector of coefficients and the right argument determines the point at which the polynomial is evaluated.)

If the coefficients *C* are defined as follows:

```
      R←0,⍳7
      R
0  1  2  3  4  5  6  7
      !R
1  1  2  6  24  120  720  5040
      C←1÷!R
      C
1  1  0.5  0.1666666667  0.04166666667  0.008333333333
      0.001388888889  0.0001984126984
```

then

```
      C POL 1
2.718253968
      C POL ¯1
0.3678571429
      (C POL 1)×C POL ¯1
0.9999291383
      C POL 2
7.380952381
```

and the polymonial is clearly an approximation to the exponential function.

The use of the computer for plotting functions is helpful in all experiments with functions. It is however, desirable to compose a slightly more complicated plotting function which will perform automatic scaling as required.

More advanced mathematical examples. The examples thus far have all addressed an elementary level of mathematics. The computer is equally useful for experiments in more advanced topics. This section will present a few brief examples of such use.

In this work, the following bits of matrix notation will be required:

1. The expression $D\rho X$ yields a matrix of dimension $D$ whose elements (in row-by-row order) are the elements of the vector $X$:

```
      D←3 4
      X←⍳12
      X
1  2  3  4  5  6  7  8  9  10  11  12
      M←D⍴X
      M

1   2   3   4
5   6   7   8
9  10  11  12

      ⍴X
12
      ⍴M
3  4
```

2. The expression $M[3;4]$ selects the element in the third row and fourth column of $M$. More generally, $M[I;J]$ selects the row(s) determined by the elements of the vector $I$ and the column(s) selected by the vector $J$. For example:

```
      M[2;3]
7
      M[1 3;1 3 4]
1   3   4
9  11  12
```

If the index $J$ is omitted, then the entire row (or rows) is (are) taken; if the index $I$ is omitted, entire columns are taken. For example:

```
      M[2;]
5  6  7  8
      M[;2 3]
 2   3
 6   7
10  11
```

3. The expressions ⌽*M* and ⊖*M* and ⊝*M* each transpose the argument about the axis indicated by the straight line in the symbol. For example:

```
      N←⌽M
      N

1  5  9
2  6 10
3  7 11
4  8 12
      ⊖M

 4  3  2  1
 8  7  6  5
12 11 10  9
      ⊝M

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

4. The expression *M*+.×*N* denotes the ordinary matrix product of *M* and *N*. For example:

```
      M+.×N

 30  70 110
 70 174 278
110 278 446
```

More generally, any pair of operators can replace the operators + and × in the foregoing expression. If *R*←*M*α.ω*N* (where α and ω stand for any pair of operators), then *R*[*I*;*J*] is equal to α/*M*[*I*;]ω*N*[;*J*]. For example:

```
      M+.=N

4 0 0
0 4 0
0 0 4
      M⌈.⌊N

4 4  4
4 8  8
4 8 12
```

The first example is from symbolic logic. For this it will be necessary to introduce the symbols for three logical functions: ∧ for and, ∨ for or, and ~ for not, and the symbol ⌊ for integer part of a number. Consider the following experiments:

```
      L←1 0 1 1 0 1
      ∧/L
0
      ~L
0 1 0 0 1 0
      ~∨/~L
0
```

Similar experiments for further values of the logical vector $L$ suggest the theorem that $\wedge/L$ is equivalent to $\sim\vee/\sim L$, a result known as De Morgan's Law.

De Morgan's Law can easily be validated for all cases occuring for a specified number of arguments, that is, for a specified value of $\rho L$. The function

```
      ∇M←TR N
[1]   M←1=2|⌊(¯1+⍳2*N)∘.÷⍳2*N-⍳N
      ∇
```

produces a $2*N$ by $N$ matrix whose rows represent all possible logical vectors of dimension $N$. For example:

```
      T←TR 3
      T

0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1

      ∧/T
0  0  0  0  0  0  0  1
      ~∨/~T
0  0  0  0  0  0  0  1
```

A similar result (i.e., De Morgan's Law) holds for certain matrix products:

```
      ⍉T

0 0 0 0 1 1 1 1
0 0 1 1 0 0 1 1
0 1 0 1 0 1 0 1

      T∧.v⍉T

0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 1
0 0 0 0 0 1 0 1
0 0 0 0 1 1 1 1
0 0 0 1 0 0 0 1
0 0 1 1 0 0 1 1
0 1 0 1 0 1 0 1
1 1 1 1 1 1 1 1

      ~(~T)v.∧~⍉T

0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 1
0 0 0 0 0 1 0 1
0 0 0 0 1 1 1 1
0 0 0 1 0 0 0 1
0 0 1 1 0 0 1 1
0 1 0 1 0 1 0 1
1 1 1 1 1 1 1 1
```

Similar experiments can be used to suggest a host of useful identities. For example:

1. The v.∧ matrix product is (like the ordinary matrix product) associative.

2. The v.∧ matrix product distributes over v.

3. The expression ≠/$L$ (exclusive-or over the vector $L$) is equivalent to the expression 2|+/$L$ (the parity check on $L$).

4. The pair ≠/$L$ and ~=/~$L$ show a duality of the type exemplified by De Morgan's Law.

A vector such as *P*←1 4 2 3 which contains all of its indices as elements is called a permutation vector. If *X* is any vector of the same dimension as *P*, then *X*[*P*] is a permutation of the components of *X*. For example:

```
      X←'ABCD'
      X[P]
ADBC
```

If *X* is also a permutation , then *X*[*P*] is a permutation:

```
      X←3 1 4 2
      X[P]
3 2 1 4
      P[X]
2 1 3 4
```

(It is clear that permutations do not commute.)

All permutations generated by a permutation *X* can be produced as follows:

```
      Q←⍳⍴X
      Q
1  2  3  4
      Q←Q[X]
      Q
3  1  4  2
      Q←Q[X]
      Q
4  3  2  1
      Q←Q[X]
      Q
2  4  1  3
      Q←Q[X]
      Q
1  2  3  4
```

Such experiments can be used to lead to such questions as cycles, parity, and groups of permutations.

Another approach to elementary group theory can be made through concrete examples of simple finite groups. Consider, for example, the matrix *M*:

```
      M←2 2ρ'∘*O⊛'
      M

∘*
O⊛
```

The operations $\phi$, $\varphi$, and $\theta$, applied to *M* produce certain transformations which clearly belong to the 8-element group of rotations of the square (including operations which take it out of the plane):

```
      φM

∘O
*⊛
      ϕM

*∘
⊛O
      θM

O⊛
∘*
```

Further experiments can be suggested to establish whether these three operations generate the entire group, whether all three are required to generate the group, what succession of operations generate the 90-degree rotation in the plane, and so forth. More complex groups can also be modelled conveniently.

## EXAMINATIONS

The use of a computer terminal for examinations has much in common with the use of a terminal for exercises. There are, however, a few points which warrant separate discussion.

Concrete problems. Giving the student a terminal to use in an examination makes it practicable to pose more concrete problems. For example, rather than ask for the method or methods to be used in finding the roots of a polynomial, one can ask the student to find the actual roots of specified polynomials.

This has the advantage of making it easy to grade the results of an examination without retreating to multiple-choice questions. More importantly, it removes the distinction between "having the right method" and "getting the right answer", an unhealthy distinction which arises from the desire to avoid judging the student on his performance of tedious detail.

Provision of tools. Certain problems involve several distinct aspects, each of comparable difficulty. In the exposition of any one aspect, it is usually desirable to treat the other aspects as solved or solvable so as to concentrate on the question at hand. Likewise, in an examination one may concentrate on one aspect of a problem by giving the student the tools for the other aspects. This can be done by saving appropriate functions and instructing the student to load them for his own use.

Suppose, for example, the problem is to find the zeros of some empirical function for which a dozen or so values have been determined by physical experiments. One approach is to fit a polynomial to the established points and then find the zeros of this polynomial. In order to concentrate on the problem of fitting a polynomial, one would provide a zero-finding program as a tool. The curve fitting problem may itself be broken into distinct parts: 1) generating a matrix of coefficients for the implied set of linear equations, 2) inverting the matrix, and 3) multiplying the vector of function values by the inverse matrix. Any one of these aspects can be singled out for attention by providing tools for the others.

Locked functions. The formal definition of any function is normally closed by typing a del ( ∇ ). If one types instead the symbol ⍫, then the function is protected and cannot be further modified or displayed in any way. Such a function becomes a "black box" whose behaviour can be determined only by experimentation.

Locked functions can be used to pose interesting examination questions. A single locked function can serve as the basis for a series of questions of increasing difficulty:

1. What are the fundamental characteristics of a function, e.g., is its commutative or associative (for a function of two arguments) or is it even or odd (for a function of one argument).

2. Identify the function as some known function.

3. Define an equivalent function.

DRILL

In administering drill, the computer has two important advantages - it is tireless and virtually infallible. The complexity of drill can range from simple checking and correction of responses, through the compilation of statistics on the timing and correctness of responses, to the use of such statistics to diagnose conceptual difficulties indicated by the responses.

Consider, for example, spelling drill administered as follows: a computer-controlled tape recorder speaks successive words chosen by the computer program, and the student responds by typing each word on a computer terminal. The following cases indicate the potential range of sophistication:

1. For each misspelled word, the correct spelling is typed by the computer, perhaps after inviting a second try.

2. Statistics are kept on the student's performance, and the drill is concentrated on those words most frequently misspelled.

3. More detailed analysis of the misspelled words may be used to isolate, and advise the student of, any general concepts (such as the rules governing "i before e") of which he appears to be ignorant.

The methods of composing drill programs are fortunately simple enough to be mastered by both students and teachers, and do not differ significantly from the methods applicable to other problems. They will be illustrated by simple drills in spelling, multiplication, and a foreign language glossary. These examples require the use of two further pieces of notation:

1. The random function ? applied to the integer argument *N* produces a random integer in the range 1 to *N*. The function extends to vector arguments in the usual way. Thus ?6 represents the roll of a die, and ?6 6 yields a two-element result representing the roll of two independent dice. For example:

```
      ?6
4
      ?6 6
2 1
      ?3 4 5 7
3 3 1 7
```

The random function is useful in making random selections among a set of questions to be presented to a student.

2. Execution of the expression *X*←⍞ causes the keyboard to unlock and await input. The actual string of characters typed is then substituted for the ⍞ (and is in this case then assigned to the variable *X*). For example:

```
      X←⍞
ACE
      X
ACE
      X←X,⍞
TYLENE
      X
ACETYLENE
```

The ⍞ (called quote-quad) is useful for requesting and accepting student responses in a drill program.

Spelling Drill. Although spelling drill will in general require the use of an audible presentation of the questions, drill in any set of words for which there exists a convenient alternative representation (such as the representations 1, 2, 3, etc., for the integers) can be done with the typewriter alone. Consider, for example, the following sequences:

```
      W←10 5⍴'ONE  TWO  THREEFOUR FIVE SIX  SEVENEIGHT'
      ⍴W
10  5
      W

ONE
TWO
THREE
FOUR
FIVE
SIX
SEVEN
EIGHT
```

```
      ∇SPELL
[1]   Y←?8
[2]   Y
[3]   X←⍞
[4]   →⍳⌊/W[Y;]=5⍴X,'    '
[5]   'THE CORRECT SPELLING IS'
[6]   W[Y;]
[7]   →1
[8]   ∇
```

```
      SPELL
2
TWO
8
EIGT
THE CORRECT SPELLING IS
EIGHT
5
FIVE
6
SIX
```

The first lines show the construction and display of a matrix whose successive rows are the spellings of the successive integers 1 through 8. The function *SPELL* is a drill; line 2 types the integer to be spelled, line 3 accepts the student's response, line 4 compares the response (with additional spaces appended if necessary to make a five-character word) with the spelling of the integer and branches to line 1 to continue if the response is correct, and lines 5 and and 6 type the appropriate message in case of error. The last section shows use of the drill.

The foregoing drill program is defective in that it never terminates. The following revision produces a program which terminates after any empty response (i.e., a carriage return alone):

```
      ∇SPELL
[8]   [3.1]
[3.1] →0×⍳0=⍴X
[3.2] ∇
```

The revised program now appears as follows:

```
      ∇SPELL[⎕]∇
    ∇ SPELL
[1]   Y←?8
[2]   Y
[3]   X←⍞
[4]   →0×⍳0=⍴X
[5]   →⍳⌊/W[Y;]=5⍴X,'     '
[6]   'THE CORRECT SPELLING IS'
[7]   W[Y;]
[8]   →1
    ∇
```

It is clear that the branch on line 4 causes termination if the number of characters in any response is zero.

Multiplication Drill. Numeric input from the terminal is requested by the symbol ⎕ (quad). Execution of the expression typed at this point is substituted for the ⎕ in the expression (and is in this case then assigned to the variable *X*). For example:

```
      X←⎕
⎕:
      4
      X
4
      Y←⎕×X
⎕:
      3
      Y
12
```

The following sequence shows the definition and use of a simple multiplication drill in which the range of the factors presented to the student is determined by the value of the argument supplied to the drill program.

```
      ∇ DRILL N
[1]     Y←?N
[2]     Y
[3]     X←⎕
[4]     →0×⍳⌊/X='STOP'
[5]     →⍳X=×/Y
[6]     'WRONG, LOOK AT IT THIS WAY AND TRY AGAIN:'
[7]     Y⍴'⎕'
[8]     →⍳⎕=×/Y
[9]     'THE CORRECT ANSWER IS:'
[10]    ×/Y
[11]    →1
      ∇
```

```
      DRILL 10 10
1 7
□:
        7
9 3
□:
        27
5 8
□:
        50
WRONG, LOOK AT IT THIS WAY AND TRY AGAIN:

□□□□□□□□
□□□□□□□□
□□□□□□□□
□□□□□□□□
□□□□□□□□
□:
       40
5  3
□:
       17
WRONG, LOOK AT IT THIS WAY AND TRY AGAIN:
□□□
□□□
□□□
□□□
□□□
□:
       18
THE CORRECT ANSWER IS:
15
3  4
□:
       12
2  5
□:
       'STOP'
```

The first six lines of the program are similar to the first six lines of the *SPELL* program; line 7 displays a rectangle whose dimensions are the two factors presented (and the student may therefore count its elements to determine the correct answer); lines 9 and 10 type out the correct answer if the second try proves incorrect.

The same program provides drill in the multiplication of three or more factors; one need only specify the range for three values in invoking the drill:

```
      DRILL 5 6 9
5  6  1
□:
      30
5  4  5
□:
      100
2  6  5
□:
      50
WRONG, LOOK AT IT THIS WAY AND TRY AGAIN:

□□□□□
□□□□□
□□□□□
□□□□□
□□□□□
□□□□□

□□□□□
□□□□□
□□□□□
□□□□□
□□□□□
□□□□□
□:
      60
2  1  9
□:
      'STOP'
```

Personalized Drill. Given a set of one or more general drill programs, a teacher can easily define drills specialized for individual students. The following shows the definition and use of three individualized drills based on the multiplication drill:

```
      ∇JOHN
[1]   DRILL 6 6
[2]   ∇
      ∇MARY
[1]   DRILL 4 12 12
[2]   ∇
      ∇SUSAN
[1]   DRILL 50 50
[2]   ∇
```

```
      JOHN
1  4
□:
      4
3  2
□:
      6
6  4
□:
      'STOP'

      SUSAN
24  48
□:
      1152
3  39
□:
      'STOP'
```

Glossary drill. Consider the following programs called *ENTER*, and *DRILL*:

```
      ∇ ENTER
[1]     F←,F
[2]     E←,E
[3]     N←⍬
[4]     →10×⍳0=⍴N
[5]     F←F,15⍴N,15⍴' '
[6]     N←⍬
[7]     E←E,15⍴N,15⍴' '
[8]     ''
[9]     →3
[10]    F←(((⍴F)÷15),15)⍴F
[11]    E←(((⍴E)÷15),15)⍴E
      ∇

      ∇ DRILL
[1]     I←?(⍴F)[1]
[2]     F[I;]
[3]     N←⍬
[4]     →0×⍳0=⍴,N
[5]     →∧/E[I;]=15⍴N,15⍴' '
[6]     'WRONG'
[7]     N←⍬
[8]     →5×⍳0≠⍴,N
[9]     'CORRECT ANSWER IS ',E[I;]
[10]    →1
      ∇
```

The program *ENTER* accepts pairs of literal entries from the keyboard, places the first of each pair in successive rows of the matrix *F* and the second of each pair in the corresponding row of the matrix *E*. The following sequence shows the setting of the matrices *E* and *F* to "empty", the use of the *ENTER* function, and the display of the resulting values of *E* and *F*:

```
      E←''
      F←''
      ENTER
MORT
DEAD

MENER
TO LEAD

HAUT
HIGH, LOUD

QUITTER
TO LEAVE

      E

DEAD
TO LEAD
HIGH, LOUD
TO LEAVE

      F

MORT
MENER
HAUT
QUITTER
```

The *DRILL* program simply selects a row of *F* at random, accepts input from the terminal, compares the input with the corresponding row of *E*, and responds with another selection if the answer is correct, with *'WRONG'* if the first try is incorrect, and with *'WRONG'* followed by the correct answer if the second try is incorrect:

```
      DRILL
MORT
DEAD
QUITTER
LEAVE
WRONG
TO LEAVE
MORT
BITE
WRONG
HIGH
WRONG

CORRECT ANSWER IS DEAD
HAUT
HIGH, LOUD
MENER
```

Complex Drill. Although the foregoing examples are simple, they illustrate the essential techniques of drill; the use of arrays of questions and expected responses, of random selection of questions, of comparisons to direct program branches to the part of the program appropriate to various conditions, and of control parameters which permit a single general drill to be specialized to a variety of particular uses. Complex drills embody the same techniques, although the arrays may be larger and more varied, the selection and comparison procedures more complicated, and the set of control parameters more elaborate.

Drill can be very effective in the teaching of basic skills such as typing, spelling and addition and multiplication tables. The potential rote of drill in the teaching of more abstract concepts is less clear, and much experimentation is needed. Convenience in experimentation requires flexbility and power in the programming tools employed.

## SYSTEM OF PROGRAMS AS FRAMEWORK OF A DISCIPLINE

If the student is encouraged to establish a library of functions by saving each of the functions he develops to solve successive exercises assigned in a course, he will find that he is in fact developing a set of tools which are applicable in the solution of later excercises. For example, in the treatment of polynomials in a course in algebra, the student might be lead to develop functions to evaluate a polynomial, to determine the zeros of a polynomial, to determine the coefficients of a product polynomial in terms of the coefficients of the factors, and to perform synthetic division. In later work, this same set of tools would be extended by functions to determine the coefficients of a polynomial to fit a given set of points, to determine the coefficients of a polynomial which yields the slope of a given polynomial, and to approximate certain functions (such as the elementary functions) by polynomials of indefinite degree.

The mark of a well developed discipline is, in fact, the existence of a well-established set of functions for use as tools. In the study of linear systems, for example, the set includes matrix product, matrix inverse, and determinant. In statistics the set includes moments and correlations, as well as the set already mentioned for linear systems.

## SUMMARY

Programming is the key to computer use, and since the required notation is a simple extension of algebra, programming ability is easily acquired. The main applications of the computer are two - expermentation, and drill.

Drills adequate for the teaching of basic skills appear to be relatively easy to construct. Effective drills for the teaching of abstract concepts are much more difficult to construct, and results in this area have been generally disappointing.

Student use of the computer for experimentation can be immediately effective, even with the use of established texts. A re-working of texts from an algorithmic viewpoint would, however, be desirable.

Printed in U.S.A. 320-0996-0

IBM

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
(USA Only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)