ED 212 479 SE 036 090

AUTHOR	Matthews, George E.
TITLE	Verbalizing Mathematics Using APL.
PUB DATE	Oct 81
NOTE	15p.; Paper presented at the Annual Meeting of the American Mathematical Association of Two Year Colleges (New Orleans, LA, October, 1981).
EDRS PRICE	MF01/PC01 Plus Postage.
DESCRIPTORS	College Mathematics; *Computer Programs; Computer Science; Educational Technology; Higher Education; *Mathematical Vocabulary; *Mathematics; *Mathematics Education; Mathematics Instruction; Problem Solving; *Programing; Proof (Mathematics)
IDENTIFIERS	*APL Programing Language; *Computer Uses in Education

ABSTRACT

The nature of "A Programing Language" (APL) is viewed as unambiguous, consistent, and powerful. It is based on the notion of functions as imperative verbs, and is used by a small but growing number of mathematicians and computer programers. Three areas of mathematical activity are addressed: calculation of arithmetic expressions, evaluation of algebraic formulas, and computation of algebraic processes. The uses of APL in each of these areas is illustrated by elementary examples. Because of its design as a language rich in primitive functions, with extensions created by operators and user-defined functions, APL is seen as a powerful tool for mathematical exposition. (MP)

VERBALIZING MATHEMATICS USING APL

George E. Matthews
Associate Professor of Mathematics
Onondaga Community College
Syracuse, New York

If mathematics is something that people do, then mathematical exposition should be rich in verb forms. However, conventional algebraic notation is better suited for describing static results than for dynamic processes.

APL (an acronym for A Programming Language) is a modern mathematical notation that is unambiguous, consistent and powerful. Furthermore, it is based on the notion of functions as imperative verbs. As such, it is an effective means of communication for both people and computers. Created by the mathematician Kenneth Iverson and published in 1962, APL now is used by a small but growing number of mathematicians and computer programmers.

This paper addresses three areas of mathematical activity: calculation of arithmetic expressions, evaluation of algebraic formulas, and computation of algebraic processes. The uses of APL in each of these areas is illustrated by elementary examples.

OVERVIEW OF APL

A few comments will suffice to characterize APL for those who are not familiar with its design. Further details can be found in works such as (Falkoff and Iverson, 1973), (Gilman and Rose, 1976), (Iverson, 1972 a and b), and (Peelle, 1979).

APL may be viewed as an alternative mathematical notation that is directly executable on machines (computers). It is structured like a "natural" algorithmic language, with functions serving as verbs, constants as its nouns and variables as its pronouns.

APL uses arrays of constants as basic data, it has a comprehensive set of primitive functions, and uses operators and user-defined functions to [illegible] the scope of the defined symbols. There is virtuall[illegible] restriction on choices of names for functions or variables in APL.

All functions in APL are treated alike, in a right-to-left, arithmetic syntax; parentheses are needed only for expressions used as left-hand inputs. Statements in APL result in either assignment of values to named storage, branching to another numbered statement, or display of computed results.

APL uses many familiar symbols from conventional algebra but regularizes their syntax and assigns special meanings for both familiar and new symbols. The consistent form of all function usage in APL results in concise expressions that are easy to comprehend once initial familiarity is achieved. The points mentioned above can best be seen through specific examples.

ARITHMETIC CALCULATIONS

The basic operations of arithmetic (addition, subtraction, multiplication, division, and exponentiation) are viewed as functions with two inputs. Any expression containing such arithmetic symbols is viewed as a directive to calculate a value. Thus 2 + 3 is 5, 3 × 4 is 12, and 2 * 5 is 32.

If an expression appears on the right side of a function symbol, it is evaluated before the left input (if any). As usual, parentheses can be used to alter this built-in-order of operations. Hence 2 + 3 × 4 is 14 and (2 + 3) × 4 is 20 and 4 × 2 + 3 is also 20.

APL adopts this simple and uniform order of operations so that the great number of functions defined in APL can be used easily, without regard to complicated rules of precedence or heavy use of parentheses. The symbols for the basic operations are also used for related single-input functions; additional useful arithmetic functions are also defined, as shown in figure 1. The single-input functions are called monadic and the functions with two arguments are called dyadic.

ARITHMETIC SYMBOLS IN APL

+	Identity,	Addition	⌈	Ceiling,	Maximum
-	Negation,	Subtraction	⌊	Floor,	Minimum
×	Signum,	Multiplication	!	Factorial,	Binomial
÷	Reciprocal,	Division	|	Magnitude,	Residue
*	Power of e,	Exponentiation	○	Pi times,	Circular, etc.
⍟	Natural log,	Logarithmic	⌹	Matrix inverse,	Matrix divide

Figure 1. Monadic and dyadic arithmetic functions.

For example, if the subtraction sign is also used for negation, then obviously the division sign may be used for reciprocals. Thus - 1 is ¯1 and ÷ 4 is 0.25. The vertical line is used for absolute value as in |¯5 which is 5 and for division remainder as in 3 | 14 which is 2. Factorial three is written ! 3 and two pi is written ○ 2. The square root of three may be represented as 3 * ÷ 2 , in a manner which easily extends to any desired root.

With the full complement of propositional (logical and relational) functions useable in APL, it is possible to make arithmetic statements without using additional notations outside APL. For example, 0 ≠ 3|14 means that 3 is not an exact divisor of 14. <u>Conditional</u> valuations can also be expressed directly as in 3 + 2 × 4 < X, which is 3 + 2 if 4 is less than X, otherwise it is just 3 . More examples of this sort will be illustrated in the sequel.

By far the greatest attribute of APL is its handling of arrays (lists, tables, etc.). All of the elementary functions extend to arrays on an element-by-element basis. Thus ÷ 1 2 4 is 1 0.5 0.25, 2 × 1 3 4 is 2 6 8, 3 2 + 4 6 is 7 8. Note that single numbers are extended as needed to match the array.

Special array-based functions are defined in APL to provide ease in generating, manipulating, and restructuring arrays; such functions are called <u>mixed</u> functions. Additionally, the use of operators, which extend the scope of the arithmetic and propositional functions, allows concise expression of simple ideas.

The arithmetic mean provides an example: If X is a list of numbers, then +/ X is the sum of these numbers and ρX is the count of these numbers. The former expression uses the <u>reduction</u> operator and the latter uses the mixed function <u>shape.</u> Thus (+/ X) ÷ ρ X is the simple arithmetic mean. See figure 2 for specific cases of the simple mean, weighted mean, and mean of selected scores.

```
⍝                    ARITHMETIC MEANS IN APL

      X ← 78 90 83 75 79
      +/X
405
      ρX
5
      (+/X) ÷ ρX
81
⍝                    SIMPLE MEAN

      SCORES ← 84 78 90 83 79
      WTS ← 2 1 1 1 1
      WTS × SCORES
168 78 90 83 79
      (+/WTS × SCORES) ÷ +/WTS
83
⍝                    WEIGHTED MEAN

      X ← 78 90 83 75 79
      HIGHEST4 ← 4 ↑ X[⍒X]
      84 , HIGHEST4
84 90 83 79 78
      (+/WTS × 84 , HIGHEST4) ÷ +/WTS
83
⍝                    ADJUSTED MEAN


⍝ FIGURE 2.  EXAMPLES OF OPERATORS AND MIXED FUNCTIONS
```

The expression for the adjusted mean shown in figure 2 could be read as "<u>SUM</u> WEIGHTS <u>TIMES</u> 84 <u>WITH</u> 4 <u>HIGHEST</u> X all <u>OVER</u> <u>SUM</u> WEIGHTS."

The underlined words represent the dynamic parts of the expression and correspond to operators (SUM), APL primitives (TIMES, WITH, OVER), and a user-defined function (HIGHEST). Appropriate definitions of APL programs for these verbs are shown in Appendix two and explained later. Also illustrated in the above expressions are the pronouns (variables) WEIGHTS and X, the nouns (constants) 84 and 4, and the use of "all" to denote the required parentheses.

ALGEBRAIC EVALUATIONS

In the arithmetic calculations just cited, the variables were merely convenient names for specific constants. More generally, variables are used to represent indeterminates (parameters or unknowns) in algebraic formulas.

In a sense, such formulas represent hypothetical statements that have meaning (value) only when certain other information is given. APL has a convenient way to represent such formulas as character strings which can be evaluated later using the built-in Execute function.

The statistical variance of a list of data, defined as the mean squared deviations from the mean, can be represented by various formulas. A literal translation of the definition into APL is easy but not very readable. Note the triply-nested parentheses shown in the first formula in figure 3, with division being the final function performed.

A simpler computational formula expresses variance as the mean of the squares minus the square of the mean. This formula has only doubly-nested parentheses, with subtraction being the final function performed.

```
⍝            EVALUATING STATISTICAL VARIANCE

        F1 ← ' (+/ (X - (+/X) ÷ ρX) * 2) ÷ ρX '

        F2 ← ' ( (+/X * 2) ÷ ρX) - ( (+/X) ÷ ρX) * 2 '

        X ← 2 4 6 8 10 12 14
        ⍎ F1
16
        ⍎ F2
16

        X ← 7 9 6 8 8
        ⍎ F1
1.04
        ⍎ F2
1.04


⍝ FIGURE 3.  EXAMPLES OF FORMULAS AS APL STRINGS
```

The formula in any case is enclosed in quotes, which mean that the symbols therein are characters without intrinsic meaning or value. The Execute function, denoted by ⍎ , has the effect of stripping off the quotes and interpreting the string as if it were a statement directly entered by the user. This system allows several formulas to be stored as named character strings for later recall and evaluation. Such strings may contain functions, constants and variables as desired. After appropriate values have been specified for the variables, the formulas can be evaluated.

A common algebraic activity is graphing functions. Figure 4 shows a manner of graphing the function (X - 3)(X - 5) on the domain of 1,2,3,4,5,6,7 . The technique involves character strings, the Execute function, the outer product operator, and indexing. The details need not concern us here, for the example is intended merely to illustrate the scope of algebraic evaluations in APL.

```
⍝                 GRAPHING ALGEBRAIC FUNCTIONS

       F ← ' (X-3) × X-5 '
       X ← 1 2 3 4 5 6 7
       ⎕ ← Y ← ⍎ F
8 3 0 ¯1 0 3 8
       (0 = ⍎ F) / X
3 5
       ⎕ ← RANGE ← 9 - ⍳10
8 7 6 5 4 3 2 1 0 ¯1
      (10 2 ⍴ ⍕RANGE), ' *' [ 1 + RANGE ∘.= Y]
8 *     *
7
6
5
4
3  *   *
2
1
0   * *
¯1   *
```

⍝ FIGURE 4. USING OUTER PRODUCT TO INDEX A SYMBOL STRING

Inasmuch as algebraic identities constitute a major interest in mathematics, it is appropriate to illustrate the use of APL in algebraic proofs. Figure 5 shows a proof that the sum of n integers starting from 1 equals half the product of n with n + 1 . Each statement on the left is equivalent to the preceding statement for the reason stated on its right. Theorems such as this can be "checked" by executing the statements for specific choices of n. As a result, an APL proof can be more convincing than a mere abstract argument.

```
⍝                     PROVING IDENTITIES IN APL

⍝      +/ ⍳N                           DEFINED FOR FINITE N
⍝      +/ ⌽ ⍳N                         +  IS  ASSOC & COMM
⍝   ( (+/⍳N) + (+/⌽⍳N) ) ÷ 2           X  ←→  (X + X) ÷ 2
⍝    (+/ (⍳N) + (⌽⍳N) ) ÷ 2            +  IS  ASSOC & COMM
⍝     (+/ N ⍴ N+1) ÷ 2                 LEMMA
⍝     ( (N+1) × N) ÷ 2                 DEF OF  ×
```

⍝ FIGURE 5. EXAMPLE OF AN APL PROOF

ALGORITHMIC COMPUTATIONS

The uses of APL illustrated in the foregoing are all for immediate execution on the computer. Nevertheless, most APL users see it as primarily a programming language, useful for writing stored programs. Much of mathematics involves algorithmic processes; APL can be invaluable for defining and exploring such procedures.

An example of an interactive program is shown in figure 6. This program illustrates the structure and use of APL programs, specifically a monadic user-defined function with explicit result. Lines 4-6 constitute a loop where the successive iterations are performed. Computer programmers will note the use of leading decision in line 4 and unconditional branching in line 6. Although APL is sometimes criticized for lacking built-in logic control structures, they can be simulated as needed or obviated by appropriate primitives and operators.

```
⍝                 NEWTON'S METHOD FOR SQUARE ROOT

      ∇ ANS ← FINDROOT N   ;G  ;EPS
[1]  EPS ← 0.0001
[2]  'GUESS'
[3]  G ← ⎕
[4]  TST: → (EPS ≥ | N - G*2) / DUN
[5]  G ← 0.5 × G + N ÷ G
[6]  → TST
[7]  DUN: ANS ← G
[8]   ∇

      FINDROOT 72.25
GUESS
⎕:
      8
8.5
```

⍝ FIGURE 6. COMPUTING SQUARE ROOT BY ITERATION

The earlier reference to a procedure for finding a weighted mean can now be more fully explained. (See figure 2). The process of adding numbers is represented in APL by the plus reduction of a list; for example, +/ 78 90 83 75 79 is the same as 78+90+83+75+79 which is 405. An alternative to using the symbols +/ is using a familiar verb such as SUM, after having defined its meaning in APL. (See Appendix two).

The verbs TIMES, WITH, OVER are actually the APL primitive functions Multiply, Join, and Divide. As shown in Appendix two, they can be given arbitrary names by means of appropriate programs. Incidentally, the underlining used for these verbs is a stylistic device and is not required.

Finally, the function HIGHEST is a dyadic user-defined function with explicit result. It uses three APL primitives Grade down, Indexing, and Take. It is the first function to be executed in the expression within parentheses.

The expression (SUM WEIGHTS TIMES 84 WITH 4 HIGHEST X) OVER SUM WEIGHTS will produce the results as shown in Appendix two, if the verbs and pronouns have been given meanings as shown. Further detail on programming uses of APL can be found in (Gilman and Rose, 1976), (Harms and Zabinski, 1977), and (LePage, 1978).

CONCLUSION

The thrust of this paper has been illustrating the dynamic aspect of mathematical expressions. Appendix three contains a comprehensive list of verb forms associated with APL primitives. Because of its design as a language rich in primitive functions, with extensions created by operators and user-defined functions, APL is a powerful tool for mathematical expositton.

APPENDIX ONE---References

Falkoff, A.D. and K.E. Iverson, "The Design of APL," IBM Journal of Research and Development. 17, No. 4, July 1973, 324-334.

Gilman, L. and A.J. Rose, APL--An Interactive Approach, 2nd ed., Revised, Wiley, New York, 1976.

Harms, E. and M.P. Zabinski, Introduction to APL and Computer Programming, Wiley, New York, 1977.

Iverson, K.E., Algebra: An Algorithmic Treatment, Addison-Wesley, Menlo Park, Cal., 1972.

Iverson, K.E., APL in Exposition, IBM Corporation, TR320-3010, Jan. 1972.

LePage, W.R., Applied APL Programming, Prentice-Hall, Englewood Cliffs, N. J., 1978.

Peelle. H.A., "Teaching Mathematics Via APL (A Programming Language)," Mathematics Teacher, 72 (1979). 97-116.

APPENDIX TWO---SOME APL PROGRAMS

```
      ∇ R ← SUM LIST
[1] R ← +/ LIST
[2]   ∇

      ∇ R ← X TIMES Y
[1] R ← X × Y
[2]   ∇

      ∇ R ← X WITH Y
[1] R ← X , Y
[2]   ∇

      ∇ R ← X OVER Y
[1] R ← X ÷ Y
[2]   ∇

      ∇ R ← N HIGHEST X
[1] R ← N ↑ X[⍒X]
[2]   ∇

      WEIGHTS ← 2 1 1 1 1
      X ← 78 90 83 75 79

      (SUM WEIGHTS TIMES 84 WITH 4 HIGHEST X) OVER SUM WEIGHTS
83
```

APPENDIX THREE---APL PRIMITIVES (Basic verbs)

<	is less than	≥	is not less than
≤	is not greater than	>	is greater than
=	is equal to	≠	is not equal to
		~	logically negate
∨	logically or	⍱	logically nor
∧	logically and	⍲	logically nand
+	identity, add	-	opposite, subtract
×	signum, multiply	÷	reciprocal, divide
*	power of e, exponentiate	⍟	natural log, logarithm
⌈	ceiling, take maximum	⌊	floor, take minimum
!	factorial, binomial	|	magnitude, take residue
○	pi times, take sine, etc.	⌹	matrix inverse, matrix divide
←	specify	∊	is a member of
↑	take from array	↓	drop from array
⍳	locate first index, count up	[]	index array
\	expand array	/	compress array
⍴	shape cf, reshape	,	ravel, join arrays
⊥	decode	⊤	encode
⍎	execute	⍕	format characters, format precision
⍒	grade down	⍋	grade up
⊖	flip, spin	⌽	reverse, rotate
⍉	transpose, section		
?	roll once, deal vector		