

AN  
SOUTHERN  
BIOGRAPHICAL  
DICTIONARY  
OF  
THE  
SOUTHERN  
STATES  
OF  
AMERICA

**A  
Source Book  
in APL**

$\rho T \phi A \leftrightarrow (\rho T) \rho A$

$\rho(\rho A) \phi A = (\rho \phi) \rho A$

$\rho A \rho B \rightarrow \rho \rho A \rho B$



---

# **A Source Book in APL**

---

***Other APL PRESS books***

*Algebra—An Algorithmic Treatment*  
*APL and Insight*  
*APL in Exposition*  
*Calculus in a New Key*  
*Elementary Analysis*  
*Introducing APL to Teachers*  
*Introduction to APL for Scientists and Engineers*  
*Proceedings of an APL Users Conference*  
*Resistive Circuit Theory*  
*Starmap*  
*1980 APL Users Meeting*

---

# A Source Book in APL

---

Papers by ADIN D. FALKOFF · KENNETH E. IVERSON

*With an Introduction by  
Eugene E. McDonnell*



APL PRESS  
Palo Alto  
1981

Iverson, Kenneth E., "Formalism in Programming Languages," *CACM* 7 Feb 1964, Copyright 1964, Association for Computing Machinery, Inc., reprinted by permission.

"Conventions Governing Order of Evaluation," from *Elementary Functions: An Algorithmic Treatment* by Kenneth E. Iverson, ©1966, Science Research Associates, Inc. Reprinted by permission of the publisher.

"Algebra as a Language," from *Algebra—An Algorithmic Treatment* by Kenneth E. Iverson, ©1977, APL Press. Reprinted by permission.

Falkoff, A. D. and K. E. Iverson, "The Design of APL," *IBM Journal of Research and Development*, Vol. 17, No. 4, July 1973, copyright 1973 by International Business Machines Corporation; reprinted with permission.

Falkoff, Adin D. and Kenneth E. Iverson, "The Evolution of APL," *SIGPLAN Notices* 13, August 1978, Copyright 1978, Association for Computing Machinery, Inc., reprinted by permission.

Iverson, Kenneth E., "Programming Style in APL," *An APL Users Meeting, Toronto, September 18, 19, 20, 1978*, I. P. Sharp Associates Limited. Reprinted by permission.

Iverson, K. E., "Notation as a Tool of Thought," *CACM* 23, August 1980, Copyright 1980, Association for Computing Machinery, Inc., reprinted by permission.

Iverson, Kenneth E., "The Inductive Method of Introducing APL," *APL Users Meeting, Toronto, October 6, 7, 8, 1980*, I. P. Sharp Associates Limited. Reprinted by permission.

International Standard Book Number: 0-917326-10-5  
APL PRESS, Palo Alto 94306

© 1981 by APL PRESS. All rights reserved.  
Printed in the United States of America

---

# *Contents*

---

<i>Introduction</i>	11
<b>1</b> <i>Formalism in Programming Languages</i>	<b>17</b>
KENNETH E. IVERSON	
<b>2</b> <i>Conventions Governing Order of Evaluation</i>	<b>29</b>
KENNETH E. IVERSON	
<b>3</b> <i>Algebra as a Language</i>	<b>35</b>
KENNETH E. IVERSON	
<b>4</b> <i>The Design of APL</i>	<b>49</b>
A. D. FALKOFF · K. E. IVERSON	
<b>5</b> <i>The Evolution of APL</i>	<b>63</b>
A. D. FALKOFF · K. E. IVERSON	
<b>6</b> <i>Programming Style in APL</i>	<b>79</b>
KENNETH E. IVERSON	
<b>7</b> <i>Notation as a Tool of Thought</i>	<b>105</b>
KENNETH E. IVERSON	
<b>8</b> <i>The Inductive Method of Introducing APL</i>	<b>131</b>
KENNETH E. IVERSON	



---

## *Introduction*

---



## *Introduction*

The purpose of this book is to provide background material for teachers and students of APL. In a course on APL the focus is necessarily on the details of the language and its use; it may not always be apparent what the purpose of a particular rule might be, nor how one piece of the language relates to the whole. This book is a collection of articles that deal with the more fundamental issues of the language. They appeared in widely scattered sources, over a period of many years, and are not always easy to find. They are arranged in the order of their appearance, so it is possible to get a sense of the development of the language from reading the articles in sequence.

The first article, *Formalism in Programming Languages*, appeared before there was an implementation. The reader who knows only contemporary APL will have to master some differences in notation in order to understand it. The effort will be repaid, however, because it condenses in a very small space some information on the properties of the scalar functions which appears nowhere else. In the discussion following the paper, R.A. Brooker asks a key question, one which has followed APL through its development:

*Why do you insist on using a notation which is a nightmare for typist and compositor and impossible to implement with punching and printing equipment currently available? What proposals have you got for overcoming this difficulty?*

The question had no good answer at the time. The best that had been proposed in-

involved transliteration rules that would have made it very difficult to work with the language. It was not until the advent of IBM's Selectric typewriter, with its replaceable printing element, that it became possible to think of developing a special APL printing element. Jean Sammet dismissed the paper in her review of it two years later by writing, "as soon as [the author] starts to defend the work on the grounds that it is currently practical, he is on very weak grounds." By the time the review appeared, however, the very impractical notation had found its implementers, and I read the review as I was sitting at a terminal connected to a 7090 system which was the time-sharing host for something called IVSYS, the immediate precursor of what would be called APL.

The second paper is connected with the transition from a pure notation to an implemented programming language. When it was written, although implementations had begun to appear, and the APL printing element had been developed, it was still not clear what was the best way to publish the language. In the book, as you can see from the selection, use was still made of boldface and italic type styles, rather than the single font imposed by the printing element. In the answer book, however, the functions were displayed in both the old style and the new, so that the user could easily see how to translate between the two.

In the third selection, *Algebra as a Language*, the case is made for the superiority of APL notation over those of conventional arithmetic and algebra. It also gives a discussion of

the analogies between teaching a mathematical notation and teaching a natural language, a note that will be heard again in the last selection. The paper makes clear that there is a larger purpose to APL than merely to give people something in which to program. What is intended is a thorough reform of the way mathematics is taught, given the existence of the computer.

The next two papers form a pair and can be discussed together. In the first, *The Design of APL*, Falkoff and Iverson give the reasons for many of the design decisions that went into APL. The occasion for the second paper, *The Evolution of APL*, was a conference on the history of programming languages. The criteria for a language to be represented at this conference were that it 1) was created and in use by 1967; 2) that it still be in use by 1977; and 3) that it had considerably influenced the field of computing. In the introduction to the proceedings, APL was described as follows:

*This language has received widespread use in the past few years, increasing from a few highly specialized mathematical uses to many people using it for quite different applications, including those in business. Its unique character set, frequent emphasis on cryptic “one-liner” programs, and its effective initial implementation as an interactive system make it important. In addition, the uniqueness of its overall approach and philosophy makes it significant.*

This quotation properly notes the success of APL in commercial areas, and also gives appropriate credit to the effectiveness of the initial implementation. One has to have lived through the trauma of early time-sharing systems to be able to appreciate how good this first APL really was. I could tell dozens of stories about how bad most early time-sharing systems were, and for each of the bad ones, I could tell a dozen stories about the good qualities of this first APL.

The last three papers have in common that they use the *direct definition* form of function definition. It is a bit early yet to say how important this concept will be, but there is beginning to be some evidence to suggest that it will have applicability in many areas of programming. At first glance, it might appear that its use would be restricted to simple mathematical functions, and might not, perhaps, be employed in large-scale programming activities. However, I have seen reasonably large report generators—involving several dozen functions—built using this form, and have seen other systems in which two or three hundred of these functions interact.

As APL enters its third decade, it promises to find a significantly larger number of users. Those who truly wish to master it should know more than just the meanings of its primitive function symbols. This book is meant to help them!

#### **A note on the origins of “APL”**

I remember quite well the day I first heard the name APL. It was the summer of 1966 and I was working in the IBM Mohansic Laboratory, a small building in Yorktown Heights, NY. The project I was working on was IBM’s first effort at developing a commercial time-sharing system, one which was called TSS. The system was showing signs of becoming incomprehensible as more and more bells and whistles were added to it. As an experiment in documentation, I had hired three summer students and given them the job of transforming the “development workbook” type of documentation we had for certain parts of the system into something more formal, namely Iverson notation, which the three students had learned while taking a course given by Ken Iverson at Fox Lane High School in Mount Kisco, NY. One of the students was Eric Iverson, Ken’s son.

As I walked by the office the three students shared, I could hear sounds of an argument going on. I poked my head in the door, and

Eric asked me, "Isn't it true that everyone knows the notation we're using is called APL?" I was sorry to have to disappoint him by confessing that I had never heard it called that. Where had he got the idea it was well known? And who had decided to call it that? In fact, why did it have to be called anything? Quite a while later I heard how it was named. When the implementation effort started in June of 1966, the documentation effort started, too. I suppose when they had to write about "it," Falkoff and Iverson realized that they would have to give "it" a name. There were probably many suggestions made at that time, but I have heard of only two. A group at SRA in Chicago which was developing instructional materials using the notation was in favor of the name "Mathlab." This did not catch on. Another suggestion was to call it "Iverson's Better Math" and then let people coin the appropriate acronym. This was deemed facetious.

Then one day Adin Falkoff walked into Ken's office and wrote "A Programming Language" on the board, and underneath it the acronym "APL." Thus it was born. It was just a week or so after this that Eric Iverson asked me his question, at a time when the name hadn't yet found its way the thirteen miles up the Taconic Parkway from IBM Research to IBM Mohansic.

There was a period of time, however, when the name was in danger of having to be changed. IBM had just gotten over the experience of having to withdraw the name NPL which it had given to its "New Programming Language," because of a conflict with the use of the same initials by Britain's National Physics Laboratory. The conflict involving APL arose when a paper appeared in the 1966 AFIPS Fall Joint Computer Conference *Proceedings*. It was by George Dodd, of General Motors Research, and was entitled *APL—a language for associative data handling in PL/I*. (PL/I was the name now given to the former NPL.) In the review of this paper that appeared in *Computing Reviews* 8, for September-October 1967 (review 12,753), Saul Rosen wrote:

*This reviewer has one suggestion that is offered quite seriously, though some readers might consider it frivolous. There already exists at least one language that is reasonably well known by its acronym APL. I refer to the language developed by Iverson for which translators and interpreters have been written on a number of computers. It would be helpful if the authors of the present article could make some minor change in the name of their processor to remove this very global ambiguity.*

George Dodd replied in a letter to the editor that appeared in *CACM* 11, for May 1968, p. 378:

*I would like to offer a rebuttal to the last paragraph of the otherwise excellent and accurate review of APL—a language for associative data handling in PL/I. . . . In the review it is pointed out that there already exists one other language known by the acronym APL; that being the language developed by Kenneth Iverson of IBM. The reviewer concludes that the name of our processor should be changed to avoid a conflict of names.*

*Before naming the language we conducted a thorough search of Computing Reviews, AFIPS Reviews, and other sources, and at that time (spring, 1966) ascertained that the APL acronym was unique. Unfortunately, Iverson's language, which is an internal IBM development project and not an announced product, has also come to be known by the same name. We feel our public reference to APL preceded Iverson's and that a more reasonable request from the reviewer would be that the name of the Iverson APL be changed.*

There was a short but fairly intense skirmish inside IBM following the George Dodd letter. I don't know all the details, but I believe the IBM branch office which handled the General

Motors account was supporting George Dodd, and the case for IBM's right to use the initials was being made by Al Rose. I don't know what became of George Dodd's processor. The issue wasn't resolved until late in 1968, and was one of the things preventing the release of APL as a product. Rose eventually won the day by making the case that Iverson had established his stake in the initials when his book *A Programming Language* was published in 1962, long before Dodd's use of the letters in 1966. The story goes that, at the final meeting to decide whether to release APL, the account representative said, "The Detroit branch office nonconcur—" at which point the vice president sitting in judgment replied, "That settles it! Branch offices don't nonconcur." And so IBM retained the use of the letters.

Curiously, in view of the National Physics Laboratory's objection to the programming language named NPL, the Applied Physics Laboratory of Johns Hopkins University never made an issue, as far as I am aware, of IBM's joint use with them of the initials APL.

There is at least one other claimant to the initials. When the IBM Philadelphia Scientific Center closed in 1974, many of the APL people there moved across the continent to the San Francisco area, to work at an IBM language development location in Palo Alto. While this was going on, one of those moving picked up a copy of the *San Francisco Chronicle* which had the headline, "APL LEAVES SAN FRANCISCO." Since he had just pulled up stakes in the Philadelphia area, he was startled to see that the same thing was about to happen again in San Francisco. On closer inspection, however, it developed that the story concerned the departure of the facilities of the steamship company, American President Lines, from the docks of San Francisco to the docks across the bay in Oakland.

Eugene E. McDonnell

*September 1981*  
*Palo Alto*

---

# *1 Formalism in Programming Languages*

---



# Formalism in Programming Languages\*

Kenneth E. Iverson

International Business Machines Corporation, Yorktown Heights, New York

## Introduction

Although the question of equivalences between algorithms expressed in the same or different languages has received some attention in the literature, the more practical question of formal identities among statements in a single language has received virtually none. The importance of such identities in theoretical work is fairly obvious. The present paper will be addressed primarily to the practical implications for a compiler.

The formal identities can be incorporated directly into a compiler, or can alternatively be used by a programmer to derive a more efficient equivalent of a program specified by an analyst. The identities cited include (1) *dualities* which permit the inclusion of only one of a dual pair as a basic operator, (2) *partitioning identities* which permit the automatic allocation of limited fast-access storage in operations on arrays, (3) *permutation identities* which permit the adoption of a processing sequence suited to the particular representation used (e.g., row list or column list of a matrix), (4) *general associativity and distributivity identities* for double operators (determined as a function of the properties of the basic operators) which permit efficient reordering of operations, (5) *transposition identities*, and (6) the automatic extension of the appropriate identities to any ad hoc operations (i.e., subroutines or procedures) defined by any user of the compiler.

The discussion will be based upon a programming language which has been presented in full elsewhere [1]. However, the relevant aspects of the language will first be summarized for reference.

---

\* Received July, 1963. Presented at a Working Conference on Mechanical Language Structures, Princeton, N. J., August 1963, sponsored by the Association for Computing Machinery, the Institute for Defense Analyses, and the Business Equipment Manufacturers Association. This work was done at Harvard University while the author was a visiting lecturer, February through June, 1963.

The problems of transliteration and syntax which commonly dominate discussions of language will here be subordinated as follows. The symbols employed will permit the immediate determination of the *class* to which each belongs; thus literals are denoted by roman type, variables are denoted by italics (lowercase, lowercase bold, and uppercase bold for scalar, vector and matrix, respectively), and operators are denoted by distinct (usually nonalphabetic) symbols. The problems of transliteration (i.e., mapping the set of symbols employed onto the smaller set provided in a computer) and of mapping positional information (such as subscripts and superscripts) onto a linear representation therefore can, and will, be subordinated to questions of the structure of an adequate language.

## The Language<sup>1</sup>

1. The left arrow " $\leftarrow$ " denotes "specification," and each<sup>2</sup> statement in the language is of the form

$$x \leftarrow \alpha$$

where  $x$  is a variable and  $\alpha$  is some function.

2. The application of any unary operator  $\circ$  to a scalar argument  $x$  is denoted by  $\circ x$ , and the application of a binary operator  $\circ$  to the arguments  $x, y$  is denoted by  $x \circ y$ . The set of basic operators and symbols is shown in Table 1. The use of the same symbol for a binary and a unary operator (e.g.,  $x \sqcup y$  for  $\min(x, y)$  and  $\sqcup x$  for largest integer not exceeding  $x$ ) produces no ambiguity and does conserve symbols.

As shown in Table 1, any relation is treated as an operator (denoted by the usual symbol for the relation) having the range *zero* and *one* (logical variables). Thus, for integers  $i$  and  $j$ , the operator " $=$ " is equivalent to the Kronecker delta.

---

<sup>1</sup> The language described here differs from that in [1] in minor details designed to further systematize and simplify its structure.

<sup>2</sup> Except for branching statements, which are not relevant to the present discussion.

TABLE 1. SYMBOLS FOR BASIC OPERATORS

UNARY		BINARY	
Operation	Symbol	Operation	Symbols
Absolute value		Arithmetic operators	+ - × ÷
Minus	-	Arithmetic relations	< ≤ = ≥ > ≠
Floor (largest integer contained)	⌊	Max, Min	∩ ∪
Ceiling (smallest integer containing)	⌈	Exponentiation ( $y^x$ )	$x \uparrow y$
Logical negation	~	Residue mod m	$m   n$
Reciprocation ( $\div x \leftrightarrow 1 \div x$ )	÷	Logical AND, OR	∧ ∨

TABLE 2. UNARY OPERATIONS DEFINED ON ARRAYS

$\nu x$	Dimension of vector $x$
$\nu A$	Row dimension of matrix $A$ (dimension of row vectors)
$\mu A$	Column dimension of matrix $A$ (dimension of column vectors)
$\ominus \oplus \odot \oslash$	Transposition of matrix about axis indicated by the straight line ( $\odot A$ is ordinary transposition of $A$ )
$\oplus$	$\oplus x$ denotes transposition of vector $x$ (reversal of order of components)
$\perp$	Base-two value of vector

3. The  $i$ th component of a vector  $x$  is denoted by  $x_i$ , the  $i$ th row vector of a matrix  $M$  by  $M^i$ , the  $j$ th column vector by  $M_j$ , and the  $(i, j)$ th element by  $M_j^i$ . A vector may be represented by a list of its components separated by commas. Thus, the statement

$$x \leftarrow 1, 2, 3, 4$$

specifies  $x$  as a vector of dimension 4 comprising the first four positive integers. In particular, catenation of two vectors  $x$  and  $y$  may be denoted by  $x, y$ .

4. Operators are extended component-by-component to arrays. Thus if  $\circ$  is any operator (unary or binary as appropriate),<sup>3</sup>

$$\begin{aligned} r \leftarrow \circ x &\leftrightarrow r_i \leftarrow \circ x_i \\ r \leftarrow x \circ y &\leftrightarrow r_i \leftarrow x_i \circ y_i \\ R \leftarrow \circ M &\leftrightarrow R_j^i \leftarrow \circ M_j^i \\ R \leftarrow M \circ N &\leftrightarrow R_j^i \leftarrow M_j^i \circ N_j^i. \end{aligned}$$

5. The order of execution of operations is determined by parentheses in the usual way and, except for intervening parentheses, operations are executed in order from *right to left*, with no priorities accorded to multiplication or other operators.

6. Certain unary operators are defined upon vectors and matrices rather than upon scalars. These appear in

<sup>3</sup> The symbol  $\leftrightarrow$  will be used to denote equivalence.

Table 2 and include the *dimension* operators  $\nu$  and  $\mu$  as well as the *transposition* operators  $\ominus, \oplus, \odot, \oslash$ , in which the symbols indicate the axis of transposition of a matrix.

7. It is convenient to provide symbols for certain constant vectors and matrices as shown in Table 3. The parenthetic expression indicating the dimension of each may be elided when it is otherwise determined by conformability with some known vector.

TABLE 3. CONSTANT VECTORS AND SQUARE MATRICES OF DIMENSION  $n$

Symbol	Designated Constant	
$\epsilon(n)$	Full vector (all 1's)	} Logical Vectors
$\epsilon^j(n)$	$j$ th unit vector (1 in position $j$ )	
$\alpha^j(n)$	Prefix vector of weight $j$ ( $j$ leading 1's)	
$\omega^j(n)$	Suffix vector of weight $j$ ( $j$ trailing 1's)	
$\nu^j(n)$	Interval vector ( $j, j+1, \dots, j+n-1$ )	
$\square(n)$	Zero matrix	} Logical Matrices
$\boxtimes(n)$	Identity matrix (1's on diagonal)	
$\boxplus(n)$	Strict upper right triangle (1's above diagonal)	
$\boxtimes(n)$	Upper right triangle (1's above and on diagonal)	
$\boxminus(n)$	Strict lower right triangle	
$\boxdot(n)$	Upper left triangle	

8. If  $\alpha(i)$  denotes one of a family of variables (e.g., scalars  $x^i$  or  $x_i$ , vectors  $x^i$  or  $X^i$  or  $X_j$ , or matrices  $X^i$ ) for  $i$  belonging to some index set  $i$ , and if  $\circ$  is a binary operator, then for any set  $s \subseteq i$ ,

$$\circ^s / \alpha(i) \leftrightarrow \alpha(s_1) \circ \alpha(s_2) \circ \dots \circ \alpha(s_{\nu s}).$$

If

$$\alpha(i) = x_i \quad \text{and} \quad s = \mathbf{1}^1(\nu x),$$

or if

$$\alpha(i) = X_i \quad \text{and} \quad s = \mathbf{1}^1(\nu X),$$

then  $s$  and  $i$  may be elided. Thus,

$$\begin{aligned} +/x &= x_1 + x_2 + \dots + x_{\nu x}, \\ \wedge/x &= x_1 \wedge x_2 \dots \wedge x_{\nu x}, \\ +/X &= X_1 + X_2 + \dots + X_{\nu X}, \text{ etc.} \end{aligned}$$

If  $\alpha(i) = X^i$  and  $s = \mathbf{1}^1(\mu X)$ , then the  $s$  and  $i$  may be elided provided that a second slash be added to distinguish this case from the preceding one. Thus,

$$\circ // X = X^1 \circ X^2 \circ \dots \circ X^{\mu X}.$$

9. If  $\alpha$  is any argument and  $\circ$  is any binary operator, then  $\circ^n / \alpha$  denotes the  $n$ th power of  $\alpha$  with respect to  $\circ$ .

Formally,

$$\bigcirc^n / \alpha \leftrightarrow \alpha \bigcirc \alpha \bigcirc \cdots \bigcirc \alpha \text{ (to } n \text{ terms).}$$

Hence  $\bigcirc^1 / \alpha = \alpha$ ,  $\bigcirc^{-1} / \alpha$  is the inverse of  $\alpha$  with respect to  $\bigcirc$ , and  $\bigcirc^0 / \alpha$  is the identity element of the operator  $\bigcirc$  (if they exist).

10. If  $\bigcirc_1$  and  $\bigcirc_2$  are binary operators, then the *matrix product*  $A_{\bigcirc_2}^{\bigcirc_1} B$  is a matrix of dimension  $\mu A \times \nu B$  defined by:

$$(A_{\bigcirc_2}^{\bigcirc_1} B)_j^i = \bigcirc_1 / A^i \bigcirc_2 B_j.$$

In particular,  $A \times B$  denotes the ordinary matrix product. Moreover, the pair  $(\bigcirc_2^1)$  behaves as a binary operator on  $A$  and  $B$  and hence may be treated as a binary operator. For example, applying the notation of part 9,  $(\times)^{-1} / A$  denotes the ordinary inverse of  $A$ .

If the post-multiplier is a vector  $x$  (i.e., a matrix of one column), the usual conventions of matrix algebra are applied:

$$(A \times x)_i = A^i \times x = + / A^i \times x.$$

Similarly,

$$(x \times B)_j = x \times B_j, \text{ and } x \times y = + / x \times y.$$

11. The *outer product* of two vectors  $x$  and  $y$  is denoted by  $x \bigcirc y$  and defined as the matrix  $M$  of dimension  $\nu x \times \nu y$  such that  $M_j^i = x_i \bigcirc y_j$ .

12. Deletion from a vector  $x$  of those components corresponding to the zeros of a logical vector  $u$  of like dimension is called *compression* and is denoted by  $u/x$ . Compression is extended to matrices both row-by-row and column-by-column as follows:

$$Y \leftarrow u/X \leftrightarrow Y^i = u/X^i$$

$$Y \leftarrow u//X \leftrightarrow Y_j = u/X_j.$$

11. If  $p$  is any vector containing only indices of  $x$ , then  $x_p$  is defined as follows:

$$y \leftarrow x_p \leftrightarrow y_i = x_{p_i}, \quad i \in \mathbf{1}(\nu p).$$

If  $p$  is a *permutation vector* (containing each of its own indices once) and if  $\nu p = \nu x$ , then  $x_p$  is a *permutation* of  $x$ .

Permutation is extended to matrices by row and by column as follows:

$$Y \leftarrow X_p \leftrightarrow Y^i = (X^i)_p$$

$$Y \leftarrow X^p \leftrightarrow Y_j = (X_j)_p.$$

12. *Left rotation* is a special case of permutation denoted by  $k \uparrow x$  and defined by

$$y \leftarrow k \uparrow x \leftrightarrow y_i = x_{(\nu x) | k+i}.$$

Right rotation is denoted by  $k \downarrow x$  and is defined analogously.

A *noncyclic left rotation (left shift)* denoted by  $\delta$  is defined as follows:

$$k \uparrow x \leftrightarrow (\sim \omega^k) \times k \uparrow x.$$

(The zero attached to the shaft of the arrow suggests that zeros are drawn into the “evacuated” positions). Similarly,

$$k \downarrow x \leftrightarrow (\sim \omega^k) \times k \downarrow x.$$

Rotations are extended to matrices in the usual way, a doubled symbol (e.g.,  $\uparrow\uparrow$ ) denoting rotation of *columns*. For example,

$$(k \uparrow\uparrow X)^i = k_i \uparrow\uparrow X^i,$$

and  $(k\epsilon) \uparrow\uparrow \square$  is a matrix with *ones* on the  $k$ th super-diagonal.<sup>4</sup>

13. Any new operator defined (e.g., by some algorithm, usually referred to as a *subroutine*) is to be denoted in accordance with Definition (2) and is extended to arrays exactly as any of the basic operators defined in the language. For example, if  $x \text{ gcd } y$  (or, better,  $x \downarrow y$ ) is used to denote the greatest common divisor of integers  $x$  and  $y$ , then  $x \downarrow y$ ,  $\downarrow / x$ , and  $X \downarrow y$  are automatically defined. Moreover, if  $n$  is a vector of integers and  $F^i$  represents the prime factorization of  $n_i$  with respect to the vector of primes  $p$  (that is,  $n = F \times p$ ), then clearly  $\downarrow / n = (\downarrow // F) \times p$ . Similarly, if  $x \uparrow y$  denotes the l.c.m. of  $x$  and  $y$ , then  $\uparrow / n = (\uparrow // F) \times p$ .

## Array Operations in a Compiler

The systematic extension of the familiar vector and matrix operations to all operators, and the introduction of the generalized matrix product, greatly increase the utility and frequency of use of array operations in programs, and therefore encourages their inclusion in the source language of any compiler. Array operations can, of course, be added to the repertoire of any source language by providing library or ad hoc subroutines for their execution. However, the general array operations spawn a host of useful identities, and these identities cannot be mechanically employed by the compiler unless the array operations are denoted in such a way that they are easily recognizable.

The following example illustrates this point. Consider the vector operation

$$x \leftarrow x + y$$

and the equivalent subroutine (expressed in ALGOL and using  $\nu x$  as a known integer):

**for**  $i = 1$  **step** 1 **until**  $\nu x$  **do** .

$x(i) := x(i) + y(i)$

<sup>4</sup> The  $\epsilon$  may be elided.



























































































































































































































































