The function of programming notation in systems design and the characteristics of a suitable language are discussed.

A brief introduction is given to a particular language (developed by the author and detailed elsewhere) which has many of the desired properties.

Application of the language is illustrated by the use of familiar examples.

Programming notation in systems design

by K. E. Iverson

In any area of design, systematic design procedures are necessarily based upon methods for the precise and formal description of the entities being designed. Because complex systems commonly embrace elements from a number of disparate disciplines (e.g., computers, programming systems, servomechanisms, accounting systems), there exists no common terminology or notation adequate for the description of an entire complex system, and hence no adequate basis for systematic "systems design".

Despite the variety in the components involved, there is an important element common to all systems design; namely, the universal concern with the *procedures* or *algorithms* executed by the system. In a fully automatic system the procedures are, by definition, explicit, and the behavior of such a system can be fully described by the explicit procedures, more commonly called *programs*. Even in semi-automatic systems a program description can be used effectively to describe the automatic portion and to isolate and identify the variables subject to specification by people or other incompletely predictable agents.

The programming notation or language used in the description of a system must be universal enough to conveniently describe programs appropriate to each of the elements embraced in a system. It must also be precise. To be truly effective in design it must further be concise and subject to formal manipulation, i.e., statements in the language must satisfy a good many significant formal identities.

It is important that a language be easy to learn, to remember, and to use. To this end, the operations incorporated should be a *systematic* extension of a relatively small number of elementary operations, the operation symbols employed should be mnemonic (i.e., each symbol should itself suggest the operation it represents as well as the relationships with other operations), and the language should be separable (i.e., it should be possible to learn and use part of the language applicable to some one area without learning the entire language).

The present paper is a brief introduction to a programming language more fully developed elsewhere.[1] It has been developed for, and already applied in, a variety of areas including microprogramming and computer organization, automatic programming systems, data representation, search and sorting procedures, matrix algebra, and symbolic logic. These and other areas of application are outlined in Reference 2 and developed more fully in the sources indicated in the bibliography.

The language

basic operations

The basic arithmetic operations provided must obviously include the four elementary arithmetic operations (to be denoted by the familiar symbols) as well as rounding to the nearest integer (up and down) and maximization and minimization. The operations of rounding a number x down and up will be called *floor* and *ceiling* and will be denoted by $\lfloor x \rfloor$ and $\lceil x \rceil$ respectively. The maximum of x and y will be denoted by $x \rceil y$ and the minimum by $x \rfloor y$.

The symbols chosen for the four operations just defined not only suggest the operations denoted, but also suggest the duality relations which hold among them, namely:

$\lfloor -x \rfloor = -\lceil x \rceil$, and

$(-x) \rfloor (-y) = -(x \rceil y)$.

These relations are easily verified for the example $x = 3.142$, $y = 2.718$ as follows:

$\lfloor -3.142 \rfloor = -4 = -\lceil 3.142 \rceil$, and

$(-3.142) \rfloor (-2.718) = -3.142 = -(3.142 \rceil 2.718)$.

The basic logical operations provided must include the familiar *and*, *or*, and *not* (*negation*). They are defined only on *logical* variables, i.e., on variables which take on only two values *true* and *false*. It is convenient to use the integers 1 and 0 to denote *true* and *false*, respectively, so that arithmetic operations can also be performed upon logical variables. For example, if x, y, and z are logical variables, then $n = x + y + z$ gives the number of them which are *true*.

The symbols used for *and*, *or* and *not* are $\wedge$, $\vee$, and an overbar, respectively. Again, the symbols reflect the important duality relation (DeMorgan's Law):

$x \wedge y = \overline{(\bar{x} \vee \bar{y})}$.

Logical variables are themselves frequently determined by the comparison of two variables x and y (not necessarily logical) to find if they satisfy some specified relation $\mathcal{R}$. This type of operation will be denoted by $(x\mathcal{R}y)$ and defined to have the value 1 or 0 according to whether the relation $\mathcal{R}$ holds or not. For example, $(2 < 3) = 1$, $(2 > 3) = 0$, and $(x > y) = \overline{(x \leq y)}$. Moreover, if x and y are themselves logical variables, then $(x \neq y)$ clearly denotes the *exclusive-or* function of x and y.

arrays

Although the number of distinct variables occurring in a complex system is normally very large, they tend to fall into a much smaller number of classes such that all members of any one class receive similar treatment. The system is rendered more tractable by grouping each class into a list or table and specifying the operations in the system as operations on entire arrays. In an accounting system, for example, a ledger is a collection of similar accounts and any "updating" process specified for the ledger implies that the process is to be applied to each account in the ledger. Similarly, the main memory of a computing system is a collection of registers, and since each register is itself a collection of characters it may be considered as a two-way array or table whose ith row corresponds to the ith memory register and whose jth column corresponds to the jth character of all registers in memory.

In mathematics, the terms *vector* and *matrix* have been given to the one-way array (list) and the two-way array (table), respectively. Since precise, convenient, and well-known conventions have long been established for vectors and matrices, these terms will be used in preference to the less formal notions of "list" and "table."

A vector will be denoted by a boldface lower case italic letter (as opposed to lightface lower case italic for a single element, or *scalar*), and a matrix will be denoted by boldface upper case italic. The ith component of a vector $\boldsymbol{x}$ is denoted by $\boldsymbol{x}_i$, the ith row of the matrix $\boldsymbol{M}$ by $\boldsymbol{M}^i$, the jth column of $\boldsymbol{M}$ by $\boldsymbol{M}_j$, and the element in the ith row and jth column by $\boldsymbol{M}^i_j$. Clearly, $\boldsymbol{M}^i$ and $\boldsymbol{M}_j$ are themselves vectors and $\boldsymbol{M}^i_j$ is a scalar.

For example, if the vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ are defined by

$$\boldsymbol{x} = (3, 6, 12, 4),$$

$$\boldsymbol{y} = (\mathrm{a}, \mathrm{e}, \mathrm{i}, \mathrm{o}, \mathrm{u}),$$

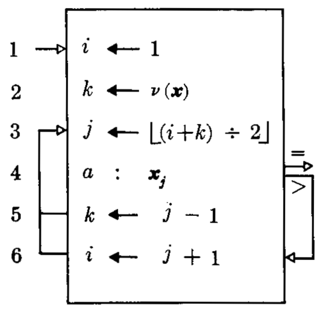
then $\boldsymbol{x}_3 = 12$, and $\boldsymbol{y}_3 = \mathrm{i}$. Moreover, if $\boldsymbol{M}$ is the logical matrix

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix},$$

then $\boldsymbol{M}^2 = (1, 0, 0, 1)$, $\boldsymbol{M}_2 = (1, 0, 0)$, and $\boldsymbol{M}^1_3 = 1$.

The *dimension* of a vector $\boldsymbol{x}$ is denoted by $\nu(\boldsymbol{x})$ and defined as the number of components of $\boldsymbol{x}$. A matrix $\boldsymbol{M}$ has two dimensions;

Figure 1 Binary search



the *row dimension* $\nu(\boldsymbol{M})$ denoting the common dimension of the row vectors $\boldsymbol{M}^i$ (i.e., the number of *columns* in $\boldsymbol{M}$), and the column dimension $\mu(\boldsymbol{M})$ denoting the dimension of the columns of $\boldsymbol{M}$. In the examples $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{M}$ of the preceding paragraph, $\nu(\boldsymbol{x}) = 4$, $\nu(\boldsymbol{y}) = 5$, $\mu(\boldsymbol{M}) = 3$, and $\nu(\boldsymbol{M}) = 4$.

The well-known binary search provides an elementary illustration of the operations introduced thus far. The objective is to determine where an argument a occurs in an ordered list of numbers, i.e., to determine the index j such that $\boldsymbol{x}_j = a$, where the vector $\boldsymbol{x}$ is the list of numbers in ascending order. The binary search procedure restricts the search to an interval $\boldsymbol{x}_i, \boldsymbol{x}_{i+1}, \cdots, \boldsymbol{x}_k$. At each stage the restriction is strengthened by comparing a with $\boldsymbol{x}_j$, where j is the index of the (approximate) midpoint of the interval, and then restricting the search to $\boldsymbol{x}_i$, $\boldsymbol{x}_{i+1}$, $\boldsymbol{x}_{j-1}$, if $a < \boldsymbol{x}_j$ or to $\boldsymbol{x}_{j+1}, \boldsymbol{x}_{j+2}, \cdots, \boldsymbol{x}_k$ if $a > x_j$.

The entire process is described by the program appearing in Figure 1. The indices i and k are first set by steps 1 and 2 to include the entire list $\boldsymbol{x}$. At each repetition of the loop beginning at step 3, the midpoint j is determined as the floor of the average of i and j. The three-way branch at step 4 terminates the process if $\boldsymbol{x}_j = a$, and respecifies either the upper limit k or the lower limit i by branching to step 5 or to step 6 as appropriate. The convenience of the notation for the dimension of a vector in setting or testing indices is apparent from step 2.

operations on arrays

The convenience of extending the addition operator to vectors in a component-by-component fashion is well known. Formally,

$$\boldsymbol{z} \leftarrow \boldsymbol{x} + \boldsymbol{y}$$

is defined (for all numerical vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ having a common dimension) by the relation $\boldsymbol{z}_i = \boldsymbol{x}_i + \boldsymbol{y}_i$, for $i = 1, 2, \cdots, \nu(\boldsymbol{x})$. In programming it is convenient to extend *all* of the basic operations on two variables in precisely the same way. For example, if $\boldsymbol{x} = (6, 3, 1, 4)$ and $\boldsymbol{y} = (1, 3, 5, 4)$, then $\boldsymbol{x} + \boldsymbol{y} = (7, 6, 6, 8)$, $\boldsymbol{x} \times \boldsymbol{y} = (6, 9, 5, 16)$, $(\boldsymbol{x} \urcorner \boldsymbol{y}) = (6, 3, 5, 4)$, $(\boldsymbol{x} > \boldsymbol{y}) = (1, 0, 0, 0)$, and $(\boldsymbol{x} > \boldsymbol{y}) \vee (\boldsymbol{x} < \boldsymbol{y}) = (1, 0, 0, 0) \vee (0, 0, 1, 0) = (1, 0, 1, 0) = (\boldsymbol{x} \neq \boldsymbol{y})$.

Each of the basic operations are similarly extended element-by-element to matrices (e.g., $\boldsymbol{X} + \boldsymbol{Y}$, $\boldsymbol{X} \times \boldsymbol{Y}$, $(\boldsymbol{X} \neq \boldsymbol{Y})$), to yield a matrix result.

The summation of all components of a vector $\boldsymbol{x}$ is frequently used and is commonly denoted by $\sum \boldsymbol{x}_i$. In order to extend this type of process (called *reduction*) to all binary operations it is necessary to employ a symbolism which incorporates the basic operator symbol (in this case $+$), thus: $+/\boldsymbol{x}$. For example, if $\boldsymbol{x} = (6, 3, 1, 4)$, and $\boldsymbol{y} = (1, 3, 5, 4)$, then $+/\boldsymbol{x} = 14$, $\times/\boldsymbol{x} = 72$, $\urcorner/\boldsymbol{x} = 6 = -(\lrcorner/(-\boldsymbol{x}))$, $\vee/(\boldsymbol{x} \neq \boldsymbol{y}) = 1$, $\wedge/(\boldsymbol{x} \neq \boldsymbol{y}) = 0$, and $+/(\boldsymbol{x} \neq \boldsymbol{y}) = 2$.

Reduction by a relation $\mathcal{R}$ is defined similarly:

$$\mathcal{R}/\boldsymbol{x} = (\cdots ((\boldsymbol{x}_1 \mathcal{R} \boldsymbol{x}_2) \mathcal{R} \boldsymbol{x}_3) \cdots \mathcal{R} \boldsymbol{x}_\nu).$$

For example, if $\boldsymbol{x} = (1, 0, 1, 1)$, then $\neq/\boldsymbol{x}$ denotes the application of the *exclusive-or* operation to $\boldsymbol{x}$, and

$$\begin{aligned}\neq/\boldsymbol{x} &= (((1 \neq 0) \neq 1) \neq 1)\\ &= ((1 \neq 1) \neq 1)\\ &= (0 \neq 1)\\ &= 1.\end{aligned}$$

The fact that an even-parity check (odd-parity codes are illegitimate) is equivalent to the *exclusive-or* may now be expressed (using the definition of residue from Table 1) as

$$\neq/\boldsymbol{x} = 2 \mid +/\boldsymbol{x}.$$

Reduction of a vector by any operator $\odot$ is extended to a matrix $\boldsymbol{M}$ in two ways: to each of the rows (denoted by $\odot/\boldsymbol{M}$ and called *row* reduction) or to each of the columns (denoted by $\odot//\boldsymbol{M}$ and called *column* reduction). Each yields a vector result.

For example, if

$$\boldsymbol{M} = \begin{bmatrix} 0, 1, 1, 0 \\ 1, 1, 1, 0 \\ 0, 0, 1, 1 \end{bmatrix}, \quad \text{and} \quad N = \begin{bmatrix} 1, 0, 1, 1 \\ 1, 1, 1, 0 \\ 1, 0, 0, 1 \end{bmatrix},$$

then $+//\boldsymbol{M} = (1, 2, 3, 1)$, $+/\boldsymbol{M} = (2, 3, 2)$, $\neq/\boldsymbol{M} = (0, 1, 0)$, and $\wedge/(\boldsymbol{M} = N) = (0, 1, 0)$. Moreover, an even-parity check on all rows of $\boldsymbol{M}$ would be denoted by $\vee/\neq/\boldsymbol{M}$, and in this example a check failure would be noted, i.e., $\vee/\neq/\boldsymbol{M} = \vee/(0, 1, 0) = 1$.

selection from arrays

Although all elements of an array may normally receive the same treatment, it is frequently necessary to select subarrays for special treatment. The selection of a single element can be indicated by a subscript (e.g., $\boldsymbol{x}_i$), but for the selection of groups of elements it is convenient to introduce the *compression operation* $\boldsymbol{u}/\boldsymbol{x}$. This is defined for an arbitrary vector $\boldsymbol{x}$ and a logical vector $\boldsymbol{u}$ of the same dimension:

$$\boldsymbol{y} \leftarrow \boldsymbol{u}/\boldsymbol{x}$$

denotes that $\boldsymbol{y}$ is obtained from $\boldsymbol{x}$ by suppressing each component $\boldsymbol{x}_i$ for which $\boldsymbol{u}_i = 0$. For example, if $\boldsymbol{x} = (\mathrm{d, e, s, i, g, n})$, and $\boldsymbol{u} = (1, 0, 0, 1, 1, 0)$, then $\boldsymbol{u}/\boldsymbol{x} = (\mathrm{d, i, g})$. Moreover, $(\boldsymbol{z} \neq \boldsymbol{y})/\boldsymbol{z}$ denotes the selection from $\boldsymbol{z}$ of those components which differ from the corresponding components of $\boldsymbol{y}$, and $+/(\boldsymbol{z} \neq \boldsymbol{y})/\boldsymbol{z}$ denotes the sum of such components. Operations are performed in order from right to left unless parentheses indicate otherwise.

Selection operations are extended to arrays in the same manner as reduction operations. Thus, if $\boldsymbol{u} = (1, 0, 1, 0)$, $\boldsymbol{v} = (1, 0, 1)$, and $\boldsymbol{M}$ and N are the matrices just employed in the examples of reduction, then

$$\boldsymbol{u}/\boldsymbol{M} = \begin{bmatrix} 0, & 1 \\ 1, & 1 \\ 0, & 1 \end{bmatrix}, \qquad \boldsymbol{v}//\boldsymbol{M} = \begin{bmatrix} 0, & 1, & 1, & 0 \\ 0, & 0, & 1, & 1 \end{bmatrix},$$

$$\text{and} \quad \boldsymbol{N}^3/\boldsymbol{M} = \begin{bmatrix} 0, & 0 \\ 1, & 0 \\ 0, & 1 \end{bmatrix}.$$

For example, if $\boldsymbol{I}$ is a 3×15 logical matrix representing the 3 index registers in a computer (e.g., the IBM® 7090), and if $\boldsymbol{t}$ is the 3-bit tag vector which selects the rows of $\boldsymbol{I}$ to be *or*ed together to produce the vector $\boldsymbol{z}$ finally used in indexing, then

$$\boldsymbol{z} = \vee//\boldsymbol{t}//\boldsymbol{I}.$$

Certain essential operations converse to compression (*mesh, mask,* and *expansion*) are easily defined in terms of the compression operator itself and are extended to matrices in the established manner (Reference 1, p. 19).

To specify fixed formats it is convenient to adopt notation for several special logical vectors, each of a specifiable dimension n. Thus $\boldsymbol{\alpha}^j(n)$ denotes a *prefix* vector of j leading 1's, $\boldsymbol{\omega}^j(n)$ a *suffix* vector of j trailing 1's, $\boldsymbol{\epsilon}^j(n)$ a *unit* vector with a 1 in position j, and $\boldsymbol{\epsilon}(n)$ a *full* vector of all 1's. Hence, $\boldsymbol{\alpha}^3(5) = (1, 1, 1, 0, 0)$, $\boldsymbol{\omega}^3(5) = (0, 0, 1, 1, 1)$, $\boldsymbol{\epsilon}^2(4) = (0, 1, 0, 0)$, and $\bar{\boldsymbol{\epsilon}}(n)$ is a zero vector. If the dimension n is clear from compatibility requirements it may be elided. Thus, if $\boldsymbol{c}$ is the 36-bit *command* register of the IBM 7090 (which contains the next command to be executed), then $\boldsymbol{\omega}^{15}/\boldsymbol{c}$ denotes the address portion.

number systems

The successive digits in the decimal representation of a number such as 1776 may be treated as the components of a vector $\boldsymbol{q} = (1, 7, 7, 6)$ and the number they represent is then the base ten value of the vector $\boldsymbol{q}$. More generally, a vector $\boldsymbol{t}$ may be evaluated in a mixed base system with radices specified by a *radix vector* $\boldsymbol{r}$. This operation will be called the *base* $\boldsymbol{r}$ *value* of $\boldsymbol{t}$ and will be denoted by $\boldsymbol{r} \perp \boldsymbol{t}$. To define it by example, consider the system of temporal units up to the day, for which $\boldsymbol{r} = (24, 60, 60)$. Then if $\boldsymbol{t} = (2, 3, 4)$ is the elapsed time in hours, minutes and seconds, $s = \boldsymbol{r} \perp \boldsymbol{t} = (2 \times 60 \times 60 + 3 \times 60 + 4) = 7384$ is the elapsed time in seconds.

In a fixed base b number system, $\boldsymbol{r} = b\boldsymbol{\epsilon}$. Hence $(10\ \boldsymbol{\epsilon}) \perp \boldsymbol{x}$ is the base 10 value of $\boldsymbol{x}$ and $(2\ \boldsymbol{\epsilon}) \perp \boldsymbol{y}$ is the base 2 value of $\boldsymbol{y}$. In the important case of base 2, elision of the radix vector $2\ \boldsymbol{\epsilon}$ is permitted. Hence if the $2^{15} \times 36$ logical matrix $\boldsymbol{M}$ is the memory of the IBM 7090 and $\boldsymbol{c}$ is the command register, then

$$\boldsymbol{s} \leftarrow \boldsymbol{M}^{\perp\boldsymbol{\omega}^{15}/\boldsymbol{c}}$$

describes the transfer of the operand to the storage register $\boldsymbol{s}$.

If y is any number (not necessarily integral), then $(y\ \boldsymbol{\epsilon}) \perp \boldsymbol{a}$ obviously denotes the polynomial in y with coefficients $\boldsymbol{a}$.

permutation

A reordering of the components of a vector $\boldsymbol{x}$ is called *permutation*. Any permutation can be specified by a *permutation vector* whose components take on the value of its indices in some order. Thus, $\boldsymbol{q} = (3, 1, 4, 2)$ and $\boldsymbol{r} = (1, 4, 5, 2, 3)$ are permutation vectors. Permutation of $\boldsymbol{x}$ by a permutation vector $\boldsymbol{p}$ is denoted by $\boldsymbol{x}_{\boldsymbol{p}}$ and defined as follows. If

$$\boldsymbol{y} \leftarrow \boldsymbol{x}_{\boldsymbol{p}}$$

then $\boldsymbol{y}_i = \boldsymbol{x}_{\boldsymbol{p}_i}$. For example, $\boldsymbol{x}_{\boldsymbol{q}} = (\boldsymbol{x}_3, \boldsymbol{x}_1, \boldsymbol{x}_4, \boldsymbol{x}_2)$. It is clear from the definition that permutation is conveniently executed by indirect addressing.

Rotation is a particularly important case of permutation which warrants special notation. Thus $k \uparrow \boldsymbol{x}$ denotes cyclic left shift by k places and $k \downarrow \boldsymbol{x}$ denotes cyclic right shift. For example, $2 \uparrow (t, e, a) = (a, t, e)$. Rotation of prefix and suffix vectors can be used to define *infix* vectors; e.g., $2 \downarrow \boldsymbol{\alpha}^3(6) = (0, 0, 1, 1, 1, 0)$, and $\boldsymbol{t} = (18 \downarrow \boldsymbol{\alpha}^3)/\boldsymbol{c}$ denotes the index tag portion of the command $\boldsymbol{c}$ in the IBM 7090.

Permutation is extended to matrices by rows $(\boldsymbol{X}_{\boldsymbol{p}})$ and by columns $(\boldsymbol{X}_{\boldsymbol{p}})$ in the established manner, as is rotation $(\boldsymbol{k} \uparrow \boldsymbol{X}$ and $\boldsymbol{h} \Uparrow \boldsymbol{X})$.

generalized matrix product

The ordinary matrix product, usually denoted by $\boldsymbol{AB}$, can be defined conveniently using the reduction operation:

$$(\boldsymbol{AB})_j^i = +/(\boldsymbol{A}^i \times \boldsymbol{B}_j).$$

To make explicit the role of the elementary operators $+$ and $\times$, this product will be written as $\boldsymbol{A} \overset{+}{\underset{\times}{}} \boldsymbol{B}$, and the definition will be extended more generally to $\boldsymbol{A} \overset{\odot_1}{\underset{\odot_2}{}} \boldsymbol{B}$, where $\odot_1$ and $\odot_2$ are any pair of binary operators.

Applications of the generalized matrix product abound: if $\boldsymbol{U}$ is a square logical matrix representing the direct connections in a network (node i is connected to node j if $\boldsymbol{U}_j^i = 1$), then $\boldsymbol{M} = \boldsymbol{U} \overset{\vee}{\underset{\wedge}{}} \boldsymbol{U}$ is the matrix of connections via paths of length *two*; if $\boldsymbol{D}$ is a *distance* matrix ($\boldsymbol{D}_j^i$ is the direct distance from city i to city j), then $\boldsymbol{T} = \boldsymbol{D} \overset{\rfloor}{\underset{+}{}} \boldsymbol{D}$ is the matrix of distances for the shortest trip of exactly two legs. The well-known identities of matrix algebra can be easily and usefully extended to operators other than $(\overset{+}{\underset{\times}{}})$.

Conclusion

In comparing this programming language with others, it is necessary to consider not only its use in description and analysis (which has been emphasized here), but also its use in the *execution* of algorithms, i.e., its use as a source language to be translated into computer code for the purpose of automatic execution.

In description and analysis (and hence in exposition), the advantages over other formal languages such as FORTRAN and ALGOL reside mainly in the conciseness, formalism, variability of level, and capacity for systematic extension.

The conciseness and its utility in the comprehension and the

Table 1 Basic operations for microprogramming

	Operation	Notation	Definition		Examples
OPERANDS	Scalar	a			$\boldsymbol{a} = (3, 4, 5, 6, 7)$, $\boldsymbol{b} = (8, 9)$, $\boldsymbol{c} = (3, 2, 1)$
	Vector	$\boldsymbol{a}$	$\boldsymbol{a} = (a_0, a_1, \cdots, a_{\nu(\boldsymbol{a})-1})$		$\boldsymbol{p} = (1, 0, 1, 0, 1)$, $\boldsymbol{q} = (1, 0, 1)$
	Matrix	$\boldsymbol{A}$	$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A}^0 \\ \boldsymbol{A}^{\mu(\boldsymbol{A})-1} \end{bmatrix} = (\boldsymbol{A}_0, \cdots \boldsymbol{A}_{\nu(\boldsymbol{A})-1})$	$\boldsymbol{A}^i$ is ith row vector; $\boldsymbol{A}_j$ is jth column vector	$\boldsymbol{A} = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \end{bmatrix}$ $\boldsymbol{P} = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}$
BASIC OPERATIONS	Floor	$k \leftarrow \lfloor x \rfloor$	$k \leq x < k + 1$	k, m, n, q integers; All basic operations are extended component-by-component to vectors and matrices, e.g.,	$\lfloor 3.14 \rfloor = 3$, $\lfloor -3.14 \rfloor = -4$
	Ceiling	$k \leftarrow \lceil x \rceil$	$k \geq x > k - 1$		$\lceil 3.14 \rceil = 4$, $\lceil -3.14 \rceil = -3$
	Residue mod m	$k \leftarrow m \mid n$	$n = mq + k$, $0 \leq k < m$		$7 \mid 19 = 5$, $7 \mid 21 = 0$, $7 \mid -3 = 4$
	And	$w \leftarrow u \wedge v$	$w = 1$ iff $u = 1$ and $v = 1$	$\boldsymbol{z} \leftarrow \boldsymbol{x} + \boldsymbol{y}$	
	Or	$w \leftarrow u \vee v$	$w = 1$ iff $u = 1$ or $v = 1$	$\boldsymbol{z} \leftarrow \boldsymbol{x} \times \boldsymbol{y}$	
	Negation	$w \leftarrow \bar{u}$	$w = 1$ iff $u = 0$	$\boldsymbol{W} \leftarrow \boldsymbol{U} \vee \boldsymbol{V}$	
	Proposition	$w \leftarrow (xRy)$	$w = 1$ iff x stands in relation R to y	$\boldsymbol{w} \leftarrow (\boldsymbol{x} \leq \boldsymbol{y})$	$(3 \geq 2) = 1$, $(5 \neq 2) = 1$, $(i = j) = \delta_{ij}$, $(u \neq v) =$ exclusive-or of u and v, $(u < v) = \bar{u} \wedge v$.
SPECIAL ARRAYS	Full vector	$\boldsymbol{w} \leftarrow \boldsymbol{\epsilon}(n)$	$w_i = 1$ (All 1's)	Of dimension $\nu(\boldsymbol{w}) = n$. The n may be omitted if clear from context	$\boldsymbol{\epsilon}(5) = (1, 1, 1, 1, 1)$, $\bar{\boldsymbol{\epsilon}} =$ zero vector
	Unit vector	$\boldsymbol{w} \leftarrow \boldsymbol{\epsilon}^j(n)$	$w_i = (i = j)$ (1 in position j)		$\boldsymbol{\epsilon}^0(5) = (1, 0, 0, 0, 0)$, $\boldsymbol{\epsilon}^3(5) = (0, 0, 0, 1, 0)$
	Prefix vector	$\boldsymbol{w} \leftarrow \boldsymbol{\alpha}^j(n)$	$w_i = (i < j)$ (j leading 1's)		$\boldsymbol{\alpha}^3(5) = (1, 1, 1, 0, 0)$, $\boldsymbol{\alpha}^2(4) = (1, 1, 0, 0)$
	Suffix vector	$\boldsymbol{w} \leftarrow \boldsymbol{\omega}^j(n)$	$w_i = (i \geq n - j)$ (j final 1's)		$\boldsymbol{\omega}^3(5) = (0, 0, 1, 1, 1)$, $j \downarrow \boldsymbol{\omega}^j = \boldsymbol{\alpha}^j$, $j \uparrow \boldsymbol{\alpha}^j = \boldsymbol{\omega}^j$
	Infix vector	$\boldsymbol{w} \leftarrow i \downarrow \boldsymbol{\alpha}^j(n)$	See Rotation (j 1's after i 0's)		$2 \downarrow \boldsymbol{\alpha}^3(9) = (0, 0, 1, 1, 1, 0, 0, 0, 0)$
	Interval vector	$\boldsymbol{k} \leftarrow \boldsymbol{\iota}^j(n)$	$k_i = i + j$ (Integers from j)		$\boldsymbol{\iota}^0(3) = (0, 1, 2)$, $\boldsymbol{\iota}^1(3) = (1, 2, 3)$, $\boldsymbol{\iota}^{-6}(4) = (-6, -5, -4, -3)$
	Full matrix	$\boldsymbol{W} \leftarrow \boldsymbol{E}(m \times n)$	$W_j^i = 1$ (All 1's)	of dimension $m \times n$ (may be omitted)	$\bar{\boldsymbol{E}}(m \times n) =$ zero matrix
	Identity matrix	$\boldsymbol{W} \leftarrow \boldsymbol{I}(m \times n)$	$W_j^i = (i = j)$ (Diagonal 1's)		

	Operation	*Notation*	*Definition*	*Examples*
SELECTION	Compression	$z \leftarrow u/x$	z obtained from x by suppressing each x_i for which $u_i = 0$	$p/a = (3, 5, 7)$, $\bar{p}/a = (4, 6)$, $\alpha^3/x = (x_0, x_1, x_2)$
	Row compression	$Z \leftarrow u/X$	$Z^i = u/X^i$	$\left\{ p/A = \begin{bmatrix} 0 & 2 & 4 \\ 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}, \quad q//A = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 & 6 \end{bmatrix} \right.$
	Column compression	$Z \leftarrow u//X$	$Z_j = u/X_j$	
	Mesh	$z \leftarrow \backslash x, u, y\backslash$	$\bar{u}/z = x$ and $u/z = y$ (Select in order from x or y as $u_i = 0$ or 1)	$\backslash b, p, c\backslash = (3, 8, 2, 9, 1)$ Extend to matrices as for compression
	Mask	$z \leftarrow /x, u, y/$	$\bar{u}/z = \bar{u}/x$ and $u/z = u/y$ ($z_i = x_i$ or y_i as $u_i = 0$ or 1)	$/a, p, A^0/ = (0, 4, 2, 6, 4)$
	Expansion	$z \leftarrow u\backslash y$	$\bar{u}/z = \bar{\epsilon}$ and $u/z = y$ (Mesh using zero vector for x)	$p\backslash c = (3, 0, 2, 0, 1)$
	Catenation	$z \leftarrow x \oplus y$	$z = (x_0, x_1, \cdots, x_{\nu(x)-1}, y_0, y_1, \cdots, y_{\nu(y)-1})$	$a \oplus b = (3, 4, 5, 6, 7, 8, 9)$, $x \oplus y = \backslash x, \omega^{\nu(y)}, y\backslash$
MISCELLANEOUS	Base 2 value	$z \leftarrow \perp u$	Value of u as a base 2 number	$\perp q = 5$, $\perp p = 21$
		$z \leftarrow \perp U$	$z_i = \perp U^i$	$\perp P = (3, 5)$
		$z \leftarrow \Vert\!\!\perp U$	$z_j = \perp U_j$	$\Vert\!\!\perp P = (1, 2, 3) = \iota^1(3)$
	Left rotation	$z \leftarrow k \uparrow x$	$z_i = x_j, \quad j = \nu(x) \mid (i + k)$ Cyclic left (right) rotation of x by k places	$2 \uparrow a = (5, 6, 7, 3, 4)$, $5 \uparrow a = a$
	Right rotation	$z \leftarrow k \downarrow x$	$z_i = x_j, \quad j = \nu(x) \mid (i - k)$	$1 \downarrow a = (7, 3, 4, 5, 6)$
	Reduction	$z \leftarrow \odot/x$	$z = (\cdots (x_0 \odot x_1) \odot x_2) \odot \cdots \odot x_{\nu-1})$ $\odot$ is any binary operator or relation	$+/p = 3$, $\times/c = 6$, $\wedge/p = 0$, $\neq/q = ((1 \neq 0) \neq 1) = (1 \neq 1) = 0 = 2 \mid +/q$
	Row reduction	$z \leftarrow \odot/X$	$z_i = \odot/X^i$	$+/A = (10, 15, 20)$, $\vee/P = (1, 1)$, $\neq/P = (0, 0)$
	Column reduction	$z \leftarrow \odot//X$	$z_j = \odot/X_j$	$+//A = (3, 6, 9, 12, 15)$, $+/(\neq//P) = 2$
	Matrix product	$z \leftarrow X \overset{\odot_1}{\underset{\odot_2}{}} Y$	$Z^i_j = \odot_1/(X^i \odot_2 Y_j)$, where $\odot_1$, $\odot_2$ are binary operations or relations	$X \overset{+}{\underset{\times}{}} Y$ is ordinary matrix product, $P \overset{+}{\underset{\times}{}} A = \begin{bmatrix} 3, 5, 7, 9, 11 \\ 2, 4, 6, 8, 10 \end{bmatrix}$ $c \overset{+}{\underset{\times}{}} A = (4, 10, 16, 22, 28)$, $P \overset{\wedge}{=} q = (0, 1)$, $x \overset{+}{\underset{\times}{}} y$ is ordinary scalar product, $b \overset{+}{\underset{\times}{}} b = 145$
	Maximum prefix	$w \leftarrow \alpha/u$	w is the maximum length prefix (suffix) in u	$\alpha/(1, 1, 0, 1, 0) = (1, 1, 0, 0, 0)$, $\omega/(1, 1, 0, 1, 0) = (0, 0, 0, 0, 0)$, $\alpha/p = (1, 0, 0, 0, 0)$, $\omega/p = (0, 0, 0, 0, 1)$,
	Maximum suffix	$w \leftarrow \omega/u$		$\alpha/\alpha^i = \alpha^i$, $(+/\alpha/(x = \bar{\epsilon})) \uparrow x = x$ left justified
	Representation	$z \leftarrow \rho(x)$	z is the vector representation of the character x	In 7090 $\rho(d) = (0, 1, 0, 1, 0, 0)$, $\rho(e) = (0, 1, 0, 1, 0, 1)$ In 8421 code, $\rho(0) = (0, 0, 0, 0)$, $\rho(1) = (0, 0, 0, 1)$

Reprinted from the Proceedings of the 1962 Fall Joint Computer Conference, Volume 22, by permission of AFIPS.

debugging of programs are both fairly obvious. The advantage of formalism (i.e., of numerous formal identities) in a programming language is not so clearly recognized. Programme 2 of Reference 3 provides an example of the use of formal identities in establishing the behavior of an algorithm; a similar treatment could easily be provided for Program 2 (matrix inversion by Gauss-Jordan) of Reference 2. Indeed, any valid algorithm for a specified process is itself an outline of a formal constructive proof of its own validity, the details being provided by the formal identities of the language in which the algorithm is presented.

The ability to describe a process at various levels of detail is an important advantage of a language. Reference 2 illustrates this ability in the specification of a computer; Programs 6.32 and 6.33 of Reference 1 illustrate it at a quite different level, involving the description of the *repeated-selection sort* in terms of tree operations and in terms of a representation of the tree suitable for execution on a computer.

The capacity for systematic extension is extremely important because of the impossibility of producing a workable language which incorporates directly all operations required in all areas of application; the best that can be hoped for is a common core of operations which can be extended in a systematic manner consistent with the core. As a simple example, consider the introduction of exponentiation. Since this is a binary operation, an operator symbol, say $\sqcap$, is adopted and is used *between* the operands; thus $y \sqcap x$ denotes x raised to the power y. Then $\boldsymbol{y} \sqcap \boldsymbol{x}$, $\boldsymbol{Y} \sqcap \boldsymbol{X}$, $\sqcap/\boldsymbol{x}$, etc., are automatically defined. As a further example, consider the adoption (in the treatment of number theory) of operators for greatest common divisor ($x \downharpoonleft y$) and for least common multiple ($x \upharpoonleft y$). Then $\downharpoonleft/\boldsymbol{x}$ is the g.c.d. of the numbers $\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots \boldsymbol{x}_\nu$. Moreover, if $\boldsymbol{p}$ is the vector of the first $\nu(\boldsymbol{p})$ primes (e.g., $\boldsymbol{p} = (2, 3, 5, 7, 11)$), and $\boldsymbol{f}$ is the vector of exponents in the prime factorization of a number n, then $n = \boldsymbol{f} \underset{\sqcap}{\times} \boldsymbol{p}$. Similarly, if $\boldsymbol{F}^i$ is the factorization of $\boldsymbol{n}_i$, then $\boldsymbol{n} = \boldsymbol{F} \underset{\sqcap}{\times} \boldsymbol{p}$, and clearly, $\downharpoonleft/\boldsymbol{n} = (\ \rfloor\ //\boldsymbol{F}) \underset{\sqcap}{\times} \boldsymbol{p}$, and $\upharpoonleft/\boldsymbol{n} = (\ \rceil\ //\boldsymbol{F}) \underset{\sqcap}{\times} \boldsymbol{p}$.

Compared to ordinary English, this notation shares with other formal languages the important advantage of being explicit. Moreover, it is rich enough to provide a description which is as straightforward as, and easily related to, the looser expression in English. For example, indirect addressing (via a table of addresses $\boldsymbol{p}$) is denoted by $\boldsymbol{x}_{\boldsymbol{p}_i}$.

In the matter of execution, the advantages of this notation in analysis and exposition are, in some areas at least, sufficient to justify its use even at the cost of a subsequent translation (to another source language for which a compiler exists) performed by a programmer. However, for direct use as a source language, two distinct problems arise: *transliteration* of a program in characters available on keyboards and printers, and *compilation.*

Because operator symbols were chosen for their mnemonic value rather than for availability, most of them require transliteration. However, because the symbols are used economically (e.g., the solidus "/" denotes compression as well as reduction, the symbol-doubling convention eliminates the need for special symbols for column operations, and the relational statement obviates special operators for *exclusive-or*, *implication*, etc.) the total number of symbols is small. (Note that the set of basic symbols employed in a language such as ALGOL includes each of the specially defined words such as IF, THEN, etc.)

Reference 4 outlines one of many possible simple transliteration schemes. The mnemonic value of the original symbols must be sacrificed to some extent in transliteration, but the transliteration need not impair the *structure* of the language—a matter of much greater moment.

The complexity of the compilation of a source language is obviously increased as the language becomes richer in basic operations, but is decreased by the adoption of a systematic structure. The generalized matrix product $X \, {}^{\odot_1}_{\odot_2} \, Y$, for example, greatly increases the power of the source language, but the compiler need produce only the same skeleton program required for the ordinary matrix product, permitting the specification of $\odot_1$ and $\odot_2$ as any of the basic operations in its repertoire. Moreover, the direct provision of array operations frequently simplifies rather than complicates the task of the compiler. For example, the operation $X \, {}^{+}_{\times} \, Y$ could be compiled so as to execute the basic arithmetic operations in any one of several orders and could therefore choose one best suited to the indexing and other facilities available. On the contrary, the use of DO statements as in FORTRAN or ALGOL, although it requires the programmer to specify *more* detail (i.e., the indexing), makes it difficult or impossible for the compiler to determine whether the particular order of execution specified by the indexing of the loops is *essential*, and hence inviolable.

In this brief exposition it has been impossible to explore many extensions of the notation such as set operations, files, general index-origins, and directed graphs and trees. Likewise, it has been impossible to include extended examples. However, a mastery of the simple operations introduced here should permit the interested designer to try the notation in his own work, referring to the papers indicated in the bibliography for extensions of the notation and for guidance from its previous use in applications similar to his own. The portion of the notation essential to microprogramming is summarized in Table 1.

A transliteration scheme for operators using the twelve Fortran symbols and alphabetics. (The twelve letters used can be distinguished either by prohibiting their use as variable symbols, or by underscoring as shown, punching the underscore as a period following the letter and printing it as an underscore, that is, "_" on the following line.)

Operator	*Symbol*
(	(
)	)
$\leftarrow$	\$
$\vee$	+
$\wedge$	*
$\bar{X}$	−X
+	@ +
$\times$	@ *
−	@ −
$\div$	$\underline{D}$
$\lceil X \rceil$	$\underline{C}$ X
$\lfloor X \rfloor$	$\underline{F}$ X
$\vert$	$\underline{R}$ $\underline{M}$
=	=
$\neq$	@ =
$>$	$\underline{G}$
$<$	@ $\underline{G}$
/	/
\	@ /
$\oplus$	,
$\perp$	$\underline{B}$
$\uparrow$	$\underline{L}$
$\downarrow$	$\underline{R}$
α^j	$\underline{A}$ J
ω^j	$\underline{W}$ J
ϵ	$\underline{E}$
ι^j	$\underline{I}$ J
μ	$\underline{M}$
ν	$\underline{N}$

ACKNOWLEDGMENT

I am indebted to Mr. A. D. Falkoff, Mr. A. L. Leiner, Professor A. G. Oettinger, and the Journal referees for helpful comments arising from their reading of the manuscript.

CITED REFERENCES

1. Iverson, Kenneth E., *A Programming Language*, Wiley, 1962.

2. Iverson, Kenneth E., "A Common Language for Hardware, Software, and Applications," *Proceedings of AFIPS Fall Joint Computer Conference,* Philadelphia, Dec., 1962.
3. Iverson, Kenneth E., "The Description of Finite Sequential Processes," *Proceedings of the Fourth London Symposium on Information Theory,* August 1960, Colin Cherry, Ed., Butterworth and Company.
4. Iverson, Kenneth E., "A Transliteration Scheme for the Keying and Printing of Microprograms" Research Note NC-79, IBM Corporation.

BIBLIOGRAPHY

- Brooks, F. P., Jr., and Iverson, K. E., *Automatic Data Processing,* Wiley (1963).
- Falkoff, A. D., "Algorithms for Parallel-Search Memories," *J. ACM,* October, 1962.
- Huzino, S., "On Some Applications of the Pushdown Store Technique," *Memoirs of the Faculty of Science,* Kyushu University, Ser. A, Volume XV, No. 1, 1961.
- Iverson, Kenneth E., "A Programming Language," *Proceedings of AFIPS Spring Joint Computer Conference,* San Francisco, May, 1962.
- Kassler, M., "Decision of a Musical System," Research Summary, *Communications of the ACM,* April 1962, page 223.
- Oettinger, A. G., "Automatic Syntactic Analysis and the Pushdown Store," *Proceedings of the Twelfth Symposium in Applied Mathematics,* April 1960, American Mathematical Society, 1961.
- Salton, G. A., "Manipulation of Trees in Information Retrieval," *Communications of the ACM,* February 1962, pp. 103–114.