SYSTEM DEVELOPMENT CORPORATION

AN APPLE TUTORIAL

MARVIN SCHAEFER

1 SEPTEMBER 1973

TM-5074/100/00

SYSTEM DEVELOPMENT CORPORATION

AN APPLE TUTORIAL

MARVIN SCHAEFER

1 SEPTEMBER 1973

PREPARED UNDER CONTRACT NO. NAS2-7635
FOR AMES RESEARCH CENTER, NATIONAL
AERONAUTICS AND SPACE ADMINISTRATION.

TM-5074/100/00

TABLE OF CONTENTS

CHAPTER ONE

INTRODUCTION

This tutorial manual illustrates the programming features of APPLE (A Parallel Programming Language).* APPLE is based on K. E. Iverson's APL (A Programming Language, Wiley, 1962), a language that uses generalized operators to concisely express mathematical algorithms on multi-dimensional data structures.

The generalized operators of APL and its successor *APL\360* permit a programmer to express manipulations of arrays almost as easily as he can express manipulations of scalars in conventional programming languages (e.g. FORTRAN, PL/I, etc.). The advantage of these operators is that when the programmer writes arithmetic expressions involving arrays, he does not have to go through the tedious process of writing nested loops to control the processing of the arrays. Instead, he is able to express the process as it conceptually occurs: in parallel on all of the elements of the arrays.

That programmers tend to think in terms of parallel processes on

* APPLE is not to be confused with the RADC assembly language bearing the same name which was produced for an associative processor.

arrays is borne out by the *APL\360* code produced by its large number of commercial users, who write their code as if it could be executed in parallel even though it is executed on a sequential computer. Many payroll programs, for example, treat the set of base salaries as a vector. Withholding taxes, voluntary deductions, and so forth, are then computed for the entire vector, at once, rather than on an employee-by-employee basis, because the same algorithm is used to process every employee.

One of the primary benefits of APL has been the elimination of unnecessary loops and bookkeeping. For example, a programmer writes loops far less frequently in APL in than in FORTRAN or PL/I. This tends to reduce the number of situations in which coding errors can be introduced into a program.

However, because *APL\360* is not a complete programming language, it does not eliminate all such situations. The only control operator in *APL\360* is the GOTO operator. The basic control operators--e.g., IF and DO--that have been included in FORTRAN and other languages dating from the 1950's are absent from APL. Paradoxically, one can write elaborately eloquent arithmetic expressions in APL but must resort to the techniques of assembly language programming in order to perform them more than once.

Dijkstra, Mills, Schorre and others have blamed a majority of programming errors on the unrestricted use of the GOTO

statement. Since one can control program flow only with GOTO's in APL, it was clear that there was a need for other control operators in APPLE if the possibility of coding errors was to be reduced significantly.

Consequently, APPLE contains such features as conditional statements, conditional expressions, operators for writing loops, and case statements. These control operators eliminated any need for the GOTO operator. Therefore, there is none in APPLE.

Other unique features of APPLE increase the clarity of exposition and simplify the coding process. These features further generalize APL's concepts and conventions for manipulating arrays and defining functions and operators.

APPLE is sufficiently extensible that all of its operators can be defined in the language itself. A formal specification of APPLE is contained in the "ILLIAC IV Language Requirements Study: Final Report," SDC document TM-5074/000/00, 31 January 1973.

CHAPTER TWO

HOW TO USE THIS TUTORIAL

In this tutorial, we assume that you have had some programming experience. We do not assume a sophisticated understanding of programming languages. The manual is self-contained, so you should be able to learn how to use APPLE by reading the descriptions and working out the examples that have been provided.

Because APPLE treats many mathematical concepts differently than do most other languages--certainly FORTRAN or PL/I--it is important that you read Chapters 3, 4, 6 and 7 closely, even though you may be familiar with many of the concepts. It is especially important that you be aware of the differences in how to subscript arrays or evaluate arithmetic expressions.

The tutorial is organized so that each chapter builds on its predecessors.

Pay close attention to the discussions on the order of evaluation in Chapters 4 and 10. The concept is easily learned, but you must understand it thoroughly in order to program in

APPLE.

APPLE has not yet been implemented on any computer. Consequently, we have not included any description of input/output or systems interface procedures. Those descriptions will be made available along with each APPLE implementation.

CHAPTER THREE

WHAT IS AN ARRAY?

The ILLIAC IV is a powerful computer. It was designed to simultaneously perform the same operation on a large number of data operands. For example, if we wanted to double each of a set of 50 numbers, the ILLIAC could double them all at once.

In a conventional programming language, such as FORTRAN or PL/I, you would have to assign a unique name to each of the 50 numbers in order to write a program that would double each of them. One simple way of assigning a unique name to each of the numbers is to declare an array that contains them. Then, each number would have a unique name consisting of the name of the array and a subscript (or index). The subscript would simply be a number in the range 1, 2,...,50.

There are at least two ways to write a program that would double each of the 50 numbers. One way is to simply write 50 assignment statements, each of which sets a specific element to twice its previous value. Another way is to write a loop that will iterate 50 times and in which each element is replaced by its double. When coding a program, this latter alternative is

preferable since it requires less writing. Even more important, writing the loop reduces the possibility of your making keypunching errors. While neither of the alternatives in this approach seems very important, consider the problem of doubling 100,000 numbers. In this case, we would have no alternative but to use an array and a loop since the program would be too long to write.

While these techniques are perfectly acceptable ways of programming in FORTRAN or PL/I on a sequential computer like an IBM/370 or a PDP-10, we may ask: Why write a loop to double 50 numbers on the ILLIAC when it only takes a couple of instructions in ILLIAC Machine Language? The answer is that these programming languages were not designed for computers like the ILLIAC, so there is no notation in the language to represent doubling all of the numbers at once since this is impossible on ordinary computers.

APPLE is not a conventional programming language. It is designed for use with computers on which you can double 50 numbers all at once. In fact, APPLE is designed to run on a "computer" where you can double 100,000 numbers at once. (Since no such computer exists, the APPLE compiler makes the ILLIAC simulate this imaginary computer. Thus, all you have to do to double the 50 numbers in APPLE, is to put them into an array and double the array.

Before we can show you how to write an APPLE program for doubling the array, we must first establish the terminology that is used to describe arrays and their properties. While all programmers are familiar with arrays, few programming languages treat them the same way.

3.1 Vectors and Matrices

A vector is a one-dimensional array of numbers. A vector is an ordered set of elements (i.e., a first element, second element, etc.), and we know how many elements there are. It does not matter how you write them down--horizontally, vertically, or diagonally; the number of elements in the vector will not change and there will still be a first element, second element, and so on. The number of elements in a vector is called its dimensionality.

Example. If V is a vector, then we represent its dimensionality by writing ρV. (ρ is the greek letter "rho.") For example, if V is the vector consisting of the four numbers (14, 3, 2, 17), then ρV is equal to 4.

A matrix is a rectangular array of numbers. Each matrix has a number of rows and a number of columns. Mathematicians can specify a particular element of a matrix by calling out, for example, the third element of the fourth row, or the fourth element of the third column. This is completely unambiguous.

They can also call out either the entire fifth row, or the sixth column, or one of the diagonals of the matrix.

A matrix has two important dimensions: the number of rows and the number of columns it contains. The dimensionality of a matrix is defined as the vector whose first element is the number of rows and whose second element is the number of columns in the matrix. For a matrix M, the dimensionality of M is written ρM.

Example. If M is the matrix

12	4	6	5
8	10	22	15

then ρM is equal to the vector (2, 4).

The dimensionality of a vector is defined to be the vector whose only element is the number of elements in the vector. Thus, we can speak of the dimension vector of a vector or matrix.

Suppose A is either a vector or a matrix. If the vector ρA consists of only one element, then A is a vector; if ρA consists of two elements, then A is a matrix. So we need a precise way of determining the number of elements in ρA to decide whether A is a vector or a matrix.

Since ρA is always a vector, it makes sense to talk about the dimensionality of ρA, i.e., to talk about the one-element vector $\rho\rho A$ whose single element is just the number of elements in the vector ρA. (Here, we write $\rho\rho A$ to mean $\rho(\rho A)$. The parentheses

are not necessary, so we omit them.) It follows that if ρρ*A* equals 1, then ρ*A* contains one element, hence *A* is a vector. If ρρ*A* equals 2, then ρ*A* contains two elements, hence *A* is a two-dimensional array (i.e., a matrix).

It is cumbersome to talk about the dimension vector of the dimension vector of an array *A*. So we define the word rank to mean the value of the unique element of the dimension vector of the dimension vector of an array, i.e., the value of ρρ*A*. Then, a vector is a rank-1 array, and a matrix is a rank-2 array.

Mathematicians speak of row vectors and column vectors. In APPLE, these are not really vectors, but matrices. This is because a row vector always has two important dimensions: the number of elements it contains and the direction in which it is written. This is also true of column vectors. In order to be consistent with the convention of listing the number of rows first, then the number of columns when we talk about the dimensionality of a matrix. The dimension vector takes the following form. For a row vector *R* the first element of ρ*R* is always 1, the number of rows in the matrix, and the second element is the number of elements in the row vector. Similarly, the first element in the dimension vector of a column vector is always the number of elements in the column vector, while the second element is always 1.

It is possible to determine the number of elements in a matrix

by looking at its dimension vector. The number of elements is equal to the product of the number of rows in the matrix and the number of columns in the matrix. That is, the number of elements in a matrix is equal to the product of the two elements in its dimension vector.

Example. If M is a matrix having 5 rows and 7 columns, then ρM equals the vector (5, 7). There are 35 elements in M, and 35 is the product of 5 and 7.

3.2 Rank-n Arrays

In APPLE, the concept of an array is generalized to an arbitrary number of dimensions. We call this number the rank of the array. For example, a rank-3 array is an arrangement of numbers along the three coordinate axes of Euclidian 3-space. That is, the elements are arranged to form the lattice points of a rectangular parallelepiped. If A were such an array, then ρA would be a vector (a, b, c), where a, b and c correspond respectively to the number of planes, rows and columns of A, and $\rho\rho A$ equals 3. Similarly, there are rank-4 arrays, rank-5 arrays, and so forth. It is easy to see that the number of elements in a rank-n array A is the product of the elements of ρA.

3.3 Empty Arrays

APPLE permits you to work with an array A for which one or more of the elements of ρA is zero. Since the product of the elements of ρA equals zero, it follows that A contains no elements.

You will occasionally have use for empty arrays. In fact, an empty array occurs in the following section.

3.4 Scalars as Rank-0 Arrays

A scalar is a number, as distinguished from a vector, matrix, quaternion, etc. A scalar corresponds to a geometric point.

In APPLE, a scalar is an array that has no dimensions whatsoever associated with it. Consequently, there can be no elements in the dimension vector ρS associated with the scalar S. This implies that $\rho\rho S$ equals 0, the number of elements in the vector ρS. Since $\rho\rho S$ is the rank of the array S, we maintain consistency by calling a scalar a rank-0 array.

CHAPTER FOUR

NOTATION FOR CONSTANTS AND VARIABLES

In APPLE, a constant is a number the value of which never changes during the execution of a program.

A variable is not a variable in the mathematical sense. Rather, a variable is the name by which you refer to a value that you wish to store someplace and access later. The value of a variable may change during the execution of a program, or it may remain constant. The significant point is that the value of a variable can vary according to your needs, but the value of a constant is always the same.

4.1 Notation for Numbers

APPLE permits the use of integral and rational numeric quantities. These numbers are called integers and floating-point numbers, respectively.

The precision of the ILLIAC permits the representation of integers n such that $-2^{48} \leq n < 2^{48}$ (i.e. integers smaller in magnitude than 281,474,976,710,656).

Floating-point numbers are rational approximations to real numbers. The representation range for a floating-point number f is $2^{-16384} \leq |f| < 2^{16383}$, where the significant part of the mantissa is correct to 48 binary figures. The floating-point representation is automatically used for those integers that cannot be represented in 48 bits.

Integers are written the same way in APPLE as they are in normal mathematics, except that the negation sign is represented by a raised bar (¯) so that it can be distinguished from the subtraction operator. Commas may not be used to separate three digit fields, because the comma is an operator that has a unique meaning in APPLE.

Example. The number 1,234 is written 1234, while -50,762 is written ¯50762.

Floating-point numbers are also written according to the normal arithmetic conventions. Here, too, the negation sign is used to represent negative numbers.

Example. Pi may be written as 3.141592653583279, while -14.337 is written ¯14.337

You may also represent numbers in scientific notation, i.e., as the product of a number and some integral power of 10. The mantissa does not need to be normalized. Here, the number is represented by writing the number, the letter *E*, and the integral power of 10 by which the number is to be multiplied.

Example. The number -47335 can be written in scientific notation as either -4.7335×10^{4}, or $-.0047335 \times 10^{7}$ or $-47335000000 \times 10^{-6}$. In APPLE, these would be, respectively, $^{-}4.7335E4$ or $^{-}.0047335E7$ or $^{-}47335000000E^{-}6$.

4.2 Notation for Variables

Since one or more values is stored in a variable, we must have a means of referring to variables. We do this by giving the variable a name.

A name consists of an alphabetic character followed by a (possibly empty) sequence of alphameric characters. An alphabetic character is either a letter or an underscored letter. An alphameric character is either a digit, an underscored digit, or an alphabetic character. A name may not contain any imbedded blanks.

Example. The following are names:

A
A
*A*1
*A*1
*A*123456*BC*

4.3 The Equivalence Symbol

APPLE uses the double-headed arrow (↔) to represent equality. This symbol is not an APPLE operator, but serves only as a meta-linguistic device. Thus, when we wish to say that the content

of the variable A is the number 3, we write $A \leftrightarrow 3$.

If we want to be more precise and insist that A contains the scalar 3, as opposed to the vector (3), we would have to specify two facts: one related to the numeric value contained in A, the other related to the rank of A. In this case, we would write:

$$A \leftrightarrow 3$$
$$\rho\rho A \leftrightarrow 0$$

If A had been the vector containing only the number 3, then we could have written either $\rho A \leftrightarrow 1$ or $\rho\rho A \leftrightarrow 1$.

CHAPTER FIVE

BASIC ARITHMETIC OPERATORS

APPLE provides the programmer with a large number of arithmetic operators. These operators are designed to operate on arrays, rather than on scalars. Some of the operators; e.g., addition, subtraction, multiplication, division, exponentiation; are common to standard languages. The remaining operators are of the type commonly found in the mathematical subroutine libraries of major programming languages.

The operators are applied to entire arrays. The multiplication operator can be used, for example, to double all of the elements of an array without your having to write a loop. It can also be employed to multiply each element of one array by the corresponding element of another array.

In this chapter, we will introduce you to each of the arithmetic operators and then explain how it works. We will subsequently describe how you form expressions involving more than one operator. In a later chapter, we will show you how to generalize some of the arithmetic operators.

5.1 The Scalar Monadic Operators

An operator is called monadic if it operates on only one argument (or operand). A scalar monadic operator is a monadic operator that is defined in terms of its effect on a scalar operand.

Since each element of an array is a scalar, a scalar monadic operator applied to an array operand A produces a resultant array B such that $\rho A \leftrightarrow \rho B$. Each element of B equals the application of the operator to the corresponding element of A. A monadic operator is written to the left of its argument.

In the remainder of this chapter, we will use the variables A, B, C, D, U, V to represent the following arrays:

$$A \leftrightarrow \begin{pmatrix} ^{-}2 & ^{-}3 & ^{-}0 \\ 7 & ^{-}4 & ^{-}1 \end{pmatrix} \qquad B \leftrightarrow \begin{pmatrix} 2 & 3 & 5 \\ 7 & 4 & 1 \end{pmatrix}$$

$$C \leftrightarrow (1.33,\ ^{-}1.33,\ 7.0,\ 0) \qquad D \leftrightarrow (2.72,\ 3.14,\ ^{-}5.8,\ 148.3)$$

$$U \leftrightarrow (1,\ 0,\ 1,\ 0) \qquad V \leftrightarrow (1,\ 1,\ 0,\ 0)$$

where

$$\rho A \leftrightarrow (2,\ 3) \quad \rho B \leftrightarrow (2,\ 3) \quad \rho C \leftrightarrow (4) \quad \rho D \leftrightarrow (4)$$
$$\rho U \leftrightarrow (4) \quad \rho V \leftrightarrow (4)$$

5.1.1 The Identity Operator

The symbol + is used to represent the identity operator. For any array A, $+A$ equals A. In symbols, we have $+A \leftrightarrow A$.

Example:

$$+A \leftrightarrow \begin{pmatrix} ¯2 & ¯3 & ¯0 \\ 7 & ¯4 & ¯1 \end{pmatrix}$$

5.1.2 The Negation Operator

The symbol - is used to represent the negation operator. (Note that - is different from the negation sign ¯, which is only used for writing negative numbers.) For any array A, every element of A is subtracted from 0.

Example:

$$-A \leftrightarrow \begin{pmatrix} 2 & ¯3 & 0 \\ ¯7 & 4 & 1 \end{pmatrix}$$

5.1.3 The Signum Operator

The symbol × is used to represent the signum operator. For a number x, signum(x) is the function whose value is: 1 if x>0, -1 if x<0, and 0 if x=0.

Example:

$$\times A \leftrightarrow \begin{pmatrix} ¯1 & 1 & ¯0 \\ 1 & ¯1 & ¯1 \end{pmatrix}$$

5.1.4 The Reciprocal Operator

The symbol : is used to represent the reciprocal operator. $:B$ is defined for all nonzero arguments, its value is 1 divided by B.

Example:

$$:B \leftrightarrow \begin{pmatrix} 0.5 & 0.3333333333 & 0.2 \\ 0.1428571429 & 0.25 & 1 \end{pmatrix}$$

5.1.5 The Exponential Operator

The symbol * is used to represent the exponential operator. For any array A, $*A$ equals e (2.7182818284...) raised to the A power. That is, $*A$ is the natural antilogarithm of A.

Example:

$$*A \leftrightarrow \begin{pmatrix} 1.353352832E^{-}1 & 2.008553692E1 & 1.000000000E0 \\ 1.096633158E3 & 1.831563889E^{-}2 & 3.678794412E^{-}1 \end{pmatrix}$$

5.1.6 The Natural Logarithm Operator

The symbol ⊛ is used to represent the natural logarithm operator. For a strictly positive argument B, $⊛B$ is the logarithm of B to the base e.

Example:

$$⊛B \leftrightarrow \begin{pmatrix} 0.6931471806 & 1.098612289 & 1.609437912 \\ 1.9459101490 & 1.386294361 & 0.000000000 \end{pmatrix}$$

5.1.7 The Floor Operator

The symbol ⌊ is used to represent the floor operator. For a number x, the floor of x is the algebraically greatest integer less than or equal to x.

Example: If C is the vector defined on page 18, then

$$\lfloor C \leftrightarrow (1, \bar{}2, 7, 0)$$

5.1.8 The Ceiling Operator

The symbol ⌈ is used to represent the ceiling operator. For a number x, the ceiling of x is the algebraically least integer greater than or equal to x.

Example:

$$\lceil C \leftrightarrow (2, \bar{}1, 7, 0)$$

5.1.9 The Absolute Value Operator

The symbol | is used to represent the absolute value operator. The absolute value of a number x is the algebraic maximum of x and -x.

Example:

$$|A \leftrightarrow \begin{pmatrix} 2 & 3 & 0 \\ 7 & 4 & 1 \end{pmatrix}$$

5.1.10 The Random Integer Operator (Roll)

The symbol ? is used to represent the random integer operator, which is better known as "roll," as in the rolling of a die. The operand must be a positive integer array. For each scalar n, the result is a normally random integer selected from the set {1,2,...,n} or {0,1,...,n-1} according as the index origin is 1 or 0, respectively. (See Section 6.1 for a discussion of Index Origin.)

Example:

$$?B \leftrightarrow \begin{pmatrix} 2 & 3 & 5 \\ 3 & 3 & 1 \end{pmatrix}$$

5.1.11 The Logical Complementation Operator (NOT)

The symbol ~ is used to represent the logical complementation operator. The operator is defined only on the set {0,1} and transforms 1 into 0 and 0 into 1.

Example:

$$\sim U \leftrightarrow (0,\ 1,\ 0,\ 1)$$

5.1.12 The Generalized Factorial Operator

The symbol ! is used to represent the generalized factorial operator. ! is not defined for negative integers. For all non-negative integers n, the result of applying this operator is n! If x is not an integer, the result of applying this operator is the gamma function applied to x+1. If n is a negative integer, n! is undefined.

Example:

```
      !C ↔ (1.188192811, ¯4.08546585, 5040, 1)
```

5.1.13 The Multiple of π Operator

The symbol ○ is used to represent the operator that multiplies its operand by π.

Example:

```
      ○C ↔ (4.178318229, ¯4.178318229, 21.99114858, 0)
```

5.1.14 Summary of Scalar Monadic Operators

The 13 scalar monadic operators are summarized in Table I.

TABLE I

SCALAR MONADIC OPERATORS

Symbol	Name	Meaning
+	Identity	$+A \leftrightarrow A$
-	Negation	$-A \leftrightarrow 0-A$
×	Signum	$\times A \leftrightarrow \begin{cases} 1 & \text{if } A>0 \\ 0 & \text{if } A=0 \\ -1 & \text{if } A<0 \end{cases}$
÷	Reciprocal	$\div A \leftrightarrow 1\div A$
*	Exponential	$*A \leftrightarrow e^A$
⍟	Natural Logarithm	$\circledast A \leftrightarrow \ln A$
⌊	Floor	Algebraically greatest integer $\leq B$
⌈	Ceiling	Algebraically least integer $\geq B$
|	Absolute Value	$\|A \leftrightarrow \begin{cases} A & \text{if } A\geq 0 \\ -A & \text{if } A\ 0 \end{cases}$
?	Random Integer (Roll)	Random Integer between *IORG* and A
~	Logical Complementation	$\sim A \leftrightarrow 1-A$ (for $A\in\{0,1\}$
!	Generalized Factorial	$!A \leftrightarrow \begin{cases} A \text{ factorial if } A \leftrightarrow \|\lfloor A \\ (\Gamma A+1) \text{ if } A\neq\lfloor A \end{cases}$
○	π Times	$\circ A \leftrightarrow \pi\times A$

5.2 The Scalar Dyadic Operators

An operator is called dyadic if it operates on two operands. A scalar dyadic operator is a dyadic operator that is defined in terms of its effect on a pair of scalar operands.

If A and B are two arrays such that $\rho A \leftrightarrow \rho B$, then the elements of A and B may be paired according to their positions in the two arrays. We say that an element from A and an element from B are corresponding elements if they share exactly the same position within their respective arrays; that is, the subscript that identifies the one element also identifies the other. The

application of a scalar dyadic operator to two such arrays A and B produces a resultant array C where $\rho C \leftrightarrow \rho A \leftrightarrow \rho B$ and an element of C corresponds to the result of applying the operator to the corresponding elements of A and B.

A dyadic operator is written between its arguments. The variables A,B,C,D,U,V used in the discussion of dyadic operators are defined on page 18.

5.2.1 The Addition Operator

The symbol + is used to represent the addition operator.

Example:

$$A+B \leftrightarrow \begin{pmatrix} 0 & 6 & 5 \\ 14 & 0 & 0 \end{pmatrix}$$

5.2.2 The Subtraction Operator

The symbol - is used to represent the subtraction operator. The argument on the right of the operator is subtracted from the argument on its left.

Example:

$$A-B \leftrightarrow \begin{pmatrix} ¯4 & 0 & ¯5 \\ 0 & ¯8 & ¯2 \end{pmatrix}$$

5.2.3 The Multiplication Operator

The symbol × is used to represent the multiplication operator.

Example:

$$A \times B \leftrightarrow \begin{pmatrix} -4 & 9 & 0 \\ 49 & -16 & -1 \end{pmatrix}$$

5.2.4 The Division Operator

The symbol ÷ is used to represent the division operator. The argument on the left of the operator is divided by the argument on its right. The operation is defined for nonzero divisors only.

Example:

$$A \div B \leftrightarrow \begin{pmatrix} -1 & 1 & 0 \\ 1 & -1 & -1 \end{pmatrix}$$

5.2.5 The Residue Operator

The symbol | is used to represent the residue operator. If m≠0 and n are numbers, there exists an integer q such that n = mq + r, where $0 \leq r < |m|$. The symbol r represents the residue of n modulo m. The definition that follows is extended to cover the case m = 0: the residue of any nonnegative n, modulo 0 is equal to n, but remains undefined for n < 0.

When we write $A|B$, we mean the residue of B modulo A.

Example:

$$A|B \leftrightarrow \begin{pmatrix} 0 & 0 & 5 \\ 0 & 0 & 0 \end{pmatrix}$$

$$C|D \leftrightarrow (0.06,\ 0.48,\ 1.2,\ 148.3)$$

5.2.6 The Minimum Operator

The symbol $\lfloor$ is used to select the algebraic minimum of its two operands.

Example:

$$A \lfloor B \leftrightarrow \begin{pmatrix} ^{-}2 & ^{-}3 & ^{-}0 \\ 7 & ^{-}4 & ^{-}1 \end{pmatrix}$$

$$C \lfloor D \leftrightarrow (1.33,\ ^{-}1.33,\ ^{-}5.8,\ 0)$$

5.2.7 The Maximum Operator

The symbol $\lceil$ is used to select the maximum of its two operands.

Example:

$$A\lceil B \leftrightarrow \begin{pmatrix} 2 & 3 & 5 \\ 7 & 4 & 1 \end{pmatrix}$$

$$C\lceil D \leftrightarrow (2.72,\ 3.14,\ 7,\ 148.3)$$

5.2.8 The Exponentiation Operator

The symbol * is used to represent the exponentiation operator. In order to raise A to the B power, you write $A*B$.

Example:

```
A*B ↔ 4.000000000E0   2.700000000E1   0.000000000E0
      8.235430000E5   2.560000000E2  ¯1.000000000E0

B*A ↔ 2.500000000E¯1  2.700000000E1   1.000000000E0
      8.235430000E5   3.906250000E¯3  1.000000000E0

D*C ↔ (3.784222315, 0.2183149959, ¯220798.4168, 1)
```

Note that $A*B$ is not always defined. For example, $A*0.5$ is the square root of A, which is defined only for nonnegative A. Of course, $^{-}32*0.2 \leftrightarrow {}^{-}2$ since $0.2 \leftrightarrow \div 5$ and $^{-}2*5 \leftrightarrow {}^{-}32$.

5.2.9 The Logarithm Operator

The symbol ⊛ is used to represent the logarithm operator. The logarithm of A to the base B is written $B⊛A$. By definition, A and B must be strictly positive and we may have $A \leftrightarrow 1$ if and only if $B \leftrightarrow 1$ The common logarithm of A is written $10⊛A$.

Example: If

$$T \leftrightarrow \begin{pmatrix} 10 & 10 & 10 \\ 10 & 10 & 10 \end{pmatrix}$$

Then

$$T⊛B \leftrightarrow \begin{pmatrix} 0.3010299957 & 0.4771212547 & 0.6989700043 \\ 0.8450980400 & 0.6020599913 & 0.0000000000 \end{pmatrix}$$

5.2.10 The Circular Function Operator

The symbol ○ is used to represent the family of operators for all of the trigonometric and hyperbolic functions, which are collectively referred to as the circular functions. This applies to the trigonometric functions since they are defined in terms of the unit circle, and to the hyperbolic functions as a consequence of the relations sinh iz = i sin z, cosh iz = i cos z, and tanh iz = i tan z, where $i^2=-1$.

A circular function is invoked by writing $A○B$, where the value of A is used to identify the particular circular function, as

follows.

The following trigonometric functions are defined for angles in radian measure:

1○*B*	is equivalent to sin *B* .
2○*B*	is equivalent to cos *B* .
3○*B*	is equivalent to tan *B* .
¯1○*B*	is equal to arcsin *B*, where 1≥|*B*
¯2○*B*	is equivalent to arccos *B*, where 1≥|*B*
¯3○*B*	is equivalent to arctan *B*.

Three functions are useful in trigonometric identities:

4○*B*	produces the principal square root of $1+B^2$,
0○*B*	produces the principal square root of $1-B^2$, where 1≥|*B*
¯4○*B*	produces the principal square root of $\bar{}1+B^2$, where 1≤|*B*.

(Here, we abused the language somewhat by writing B^2 to mean *B**2. This was done to avoid confronting you with a complicated equivalence like ¯4○*B* ↔ (¯1+*B**2)*0.5 at this early stage. See Section 5.5 for a discussion of the priority of operators and APPLE's bracketing conventions.)

The hyperbolic functions:

5○*B*	is equivalent to sinh *B*
6○*B*	is equivalent to cosh *B*
7○*B*	is equivalent to tanh *B*
¯5○*B*	is equivalent to arcsinh *B*
¯6○*B*	is equivalent to arccosh *B*, where *B*≥1
¯7○*B*	is equivalent to arctanh *B*, where 1>|*B*.

5.2.11 The Logical Conjunction Operator (AND)

The symbol ∧ is used to represent the logical conjunction operator, AND. This operator is defined only on the set {0,1}

and is completely defined by its action on the vectors U and V (see Section 5.1):

$$U \wedge V \leftrightarrow (1, 0, 0, 0)$$

5.2.12 The Logical Disjunction Operator (OR)

The symbol $\vee$ is used to represent the logical disjunction operator, OR. This operator is defined only on the set {0, 1} and is completely defined by its action on the vectors U and V :

$$U \vee V \leftrightarrow (1, 1, 1, 0)$$

5.2.13 The Exclusive Disjunction Operator (Not Equal)

The symbol $\neq$ is used to represent the exclusive disjunction operator. This operator's domain is extended to the set of real numbers; its range is {0, 1}. $A \neq B \leftrightarrow 1$ if and only if A and B are unequal.

Example:

$$A \neq B \leftrightarrow \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

$$U \neq V \leftrightarrow (0,1,1,0)$$

5.2.14 The Equality Operator

The symbol = is used to represent the equality operator. The domain of this operator is the set of real numbers, while its range is {0,1}. $A=B \leftrightarrow 1$ if and only if A and B are equal.

Example:

$$A=B \leftrightarrow \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

$$U=V \leftrightarrow (1,0,0)$$

5.2.15 The NAND Operator

The symbol $\barwedge$ is used to represent the NAND operator. NAND is defined to be the logical complement of AND. The domain and range of NAND are {0, 1}. NAND is defined by its action on the vectors U and V:

$$U \barwedge V \leftrightarrow (0, 1, 1, 1)$$

5.2.16 The NOR Operator

The symbol $\barvee$ is used to represent the NOR operator. NOR is defined to be the logical complement of OR. The domain and range of NOR is {0, 1}. NOR is defined by its action on the vectors U and V:

$$U \barvee V \leftrightarrow (0, 0, 0, 1)$$

5.2.17 The Less-Than Operator

The symbol $<$ is used to represent the less-than operator. It maps the reals onto {0, 1}, $A<B \leftrightarrow 1$ if and only if A is less than B.

Example:

$$A<B \leftrightarrow \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

5.2.18 The Less-Than-or-Equal Operator

The symbol $\leq$ is used to represent the less-than-or-equal operator. It maps the reals onto {0,1}, $A \leq B \leftrightarrow 1$ if and only if A is not greater than B.

Example:

$$A \leq B \leftrightarrow \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

5.2.19 The Greater-Than-or-Equal Operator

The symbol $\geq$ is used to represent the greater-than-or-equal operator. It maps the reals onto {0, 1}. $A \geq B \leftrightarrow 1$ if and only if A is not less than B.

Example:

$$A \geq B \leftrightarrow \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

5.2.20 The Greater-Than Operator

The symbol > is used to represent the greater-than operator. It maps the reals onto {0, 1}, $A>B \leftrightarrow 1$ if and only if A is greater than B.

Example:

$$A > B \leftrightarrow \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

5.2.21 The Generalized Combination Operator

The symbol ! is used to represent the generalized combination operator. $A!B$ is the number of combinations of B objects taken A at a time. If a and b are nonnegative integers, the number of combinations of b objects taken a at a time is given by $C(b,a)=b!/a!(b-a)!$ The generalized combination operator uses almost the same formula, but replaces the factorial operator with the monadic generalized factorial operator. Consequently, the generalized combination operator is defined for all arguments for which the generalized factorial operator is defined.

Example: Let

$X \leftrightarrow (2, 3, 4, {}^{-}2.1, 1, 0)$

$Y \leftrightarrow (5, 3, 2, 6.5, 8, 8)$

Then

$X!Y \leftrightarrow (10,1,0,0.001295385336,8,1)$

5.2.22 The Random-Selection-Without-Replacement Operator (Deal)

The symbol $?$ is used to represent the random-selection-without-replacement operator. The result of writing $A?B$ (where $A \leq B$) is a vector R such that $\rho R \leftrightarrow A$ and the elements of R are randomly selected without replacement from the set {1, 2,..., b} or {0, 1,..., b-1} according as the index origin is 1 or 0, respectively. (See Section 6.1 for a discussion on index origin.) The operator $?$ is defined only for nonnegative scalar integer arguments. The operator can be used to simulate the dealing of bridge hands, for example.

Examples:

$6?8 \leftrightarrow (8,7,3,2,4,5)$

$6?8 \leftrightarrow (8,5,2,3,1,4)$

$13?52 \leftrightarrow (25,47,18,29,15,5,11,34,42,12,49,16,10)$

5.2.23 Summary of Scalar Dyadic Operators

The 36 scalar dyadic operators are summarized in Table II (see next page).

TABLE II
SCALAR DYADIC OPERATORS

Symbol	Name	Meaning
+	Addition	$A+B$
-	Subtraction	$A-B$
×	Multiplication	$A\times B$
÷	Division	$A\div B$
|	Residue	$A\mid B \leftrightarrow B(\text{mod } A)$
⌊	Minimum	$A\lfloor B \leftrightarrow \min\{A, B\}$
⌈	Maximum	$A\lceil B \leftrightarrow \max\{A, B\}$
*	Exponentiation	$A*B \leftrightarrow A^B$
⍟	Logarithm	$A\circledast B \leftrightarrow \log_A B$

__○__	Circular	Domain
¯7○B	arctanh B	1>|B
¯6○B	arccosh B	B≥1
¯5○B	arcsinh B	
¯4○B	(¯1+B*2)*.5	1≤|B
¯3○B	arctan B	
¯2○B	arccos B	1≥|B
¯1○B	arcsin B	1≥|B
0○B	(1-B*2)*.5	1≥|B
1○B	sin B	
2○B	cos B	
3○B	tan B	
4○B	(1+B*2)*.5	
5○B	sinh B	
6○B	cosh B	
7○B	tanh B	

∧	AND	$A\wedge B$
∨	OR	$A\vee B$
≠	Exclusive OR	$A\neq B \leftrightarrow (A\vee B)\wedge A\barwedge B$ if $A, B \in \{0,1\}$
	Inequality	$A\neq B$
=	Equality	$A=B$
⍲	NAND	$A\barwedge B \leftrightarrow \sim A\wedge B$
⍱	NOR	$A\barvee B \leftrightarrow \sim A\vee B$
<	Less Than	$A<B$
≤	Less Than Or Equal	$A\leq B$
≥	Greater Than Or Equal	$A\geq B$
>	Greater Than	$A>B$
!	Generalized Combination	$A!B \leftrightarrow (!B)\div(!A)\times!B-A$
?	Deal	

5.3 Right and Left Identities

Suppose that the symbol * is some scalar dyadic operator. If there is a number L such that $L*B \leftrightarrow B$ for every value of B, L is called a left identity of *. Similarly, if there is a number R such that $A*R \leftrightarrow A$ for every value of A, then R is called a right identity of *. If * has both a right identity R and a left identity L, it follows from elementary algebra that $R \leftrightarrow L$.

Table III summarizes the identity elements of the 36 scalar dyadic operators.

TABLE III
IDENTITY ELEMENTS OF SCALAR DYADIC OPERATORS

Operator	Left Identity	Right Identity
+	0	0
-	none	0
×	1	1
÷	none	1
*	none	1
⊛	none	none
|	0	none
○	none	none
∨	0	0
∧	1	1
⍲	none	none
⍱	none	none
!	1	none
⌈	$-\infty$	$-\infty$
⌊	∞	∞
?	*IORG*	*IORG*

The following identity elements apply only to the domain {0,1}:

>	none	0
≥	none	1
<	0	none
≤	1	none
=	1	1
≠	0	0

5.4 Right-Associative Operators

An operator $*$ is associative if, for any A, B, C, we always have $A*(B*C) \leftrightarrow (A*B)*C$. For such an operator, there is never any ambiguity; you can write $A*B*C$ and everybody knows what you mean.

However, not all operators are associative. For example, the subtraction operator is not associative since, e.g., (5-4)-3=-2, while 5-(4-3)=4. (The first interpretation is called a left

association; the second is a right association.) It is ambiguous to write 5-4-3, since it is reasonable to interpret this expression with either grouping of terms.

Before we tell you which interpretation APPLE makes, let us look at a slightly more involved expression involving subtraction, say a-b-c-d-e-f-g.

First, let us look at the parsing ((((a-b)-c)-d)-e)-f)-g. Since -b=(-1)b, we have

$$\begin{aligned}((((a-b)-c)-d)-e)-f)-g &= ((((a+(-1)b)+(-1)c)+(-1)d)+\\ &\quad +(-1)e)+(-1)f)+(-1)g\\ &= a+(-1)(b+c+d+e+f+g)\\ &= a-(b+c+d+e+f+g)\end{aligned}$$

since addition is associative. We see that this parsing is equivalent to subtracting the sum of all the other terms from the first term. This parsing is called a left-associative parsing.

If we had used the other parsing, we would have

$$\begin{aligned}a-(b-(c-(d-(e-(f-g))))) &= (a-b)+c-(d-(e-(f-g)))\\ &= (a-b)+(c-d)+e-(f-g)\\ &= (a-b)+(c-d)+(e-f)+g\\ &= (a+c+e+g)-(b+d+f)\end{aligned}$$

That is, you take the sum of the first, third, ... terms and subtract the sum of the second, fourth, ... terms. This is called a right-associative parsing. In a sense, the right-associative parsing of a-b-c-d-e-f-g is more interesting than the left-associative parsing, a-(b+c+d+e+f+g). Division and exponentiation are also nonassociative operations. They are

also more interesting when given the right-associative interpretation than when given the left-associative interpretation. For example, with left-association $A*B*C*D*E$ is just $A*(B\times C\times D\times E)$. Under right-association, it is equal to the more familiar $A*(B*(C*(D*E)))$ from algebra.

Since right-associative parses are generally more interesting for nonassociative operators than left-associative parses, all APPLE operators are treated as right-associative operators. This applies not only to expressions involving repetitions of the same operator, but to expressions involving mixtures of APPLE operators as you will see in the next section.

5.5 Bracketing Conventions and Operator Priorities

Consider the expression 2+3×4. The rules of algebra say that this expression evaluates to 14, i.e., to 2+12. This is because algebra assigns a higher priority to the multiplication operator than to the addition operator. If you had wanted this expression to evaluate to 20, you would have had to parenthesize the quantity you wanted evaluated first, writing (2+3)×4 instead. (Of course, you would have removed any possible ambiguity by writing 2+(3×4) when you wanted the expression to evaluate to 14, but this is not necessary when you know the operator priorities.)

With a few exceptions, most programming languages follow the

standard operator priorities of algebra. However, there are some differences in the way some languages treat an expression like 12÷4×3. In some languages, the result is 12+12, while the expression evaluates to 3×3 in others.

So far we have covered 49 APPLE operators. It would be difficult for anyone to remember the relative priorities between such a large number of operators. Many of the assigned priorities would appear artificial. There are approximately one hundred operators in APPLE, hence the problem is non-trivial. So in order to simplify the problems of learning APPLE, there are no operator priorities whatever in the language. Instead, the right-associative parsing convention is extended to expressions involving a mixture of operators. If you want an operator to take priority over some other operator, all you have to do is parenthesize that operator and its operands.

Returning to the expression 2+3×4, we see that it is equivalent to 2+(3×4) , that is, 14. But 4×3+2 is equivalent to 4×(3+2) or 20. Hence, expressions are not necessarily commutative in APPLE. If you wanted to have 4×3+2 equal to 14, you would have to write either (4×3)+2 or 2+3×4.

Consider 12÷4×3. The right-associativity of ÷ and × means that the expression is equivalent to 12÷(4×3), i.e. 1 . The expression means that 12 is to be divided by whatever is on the right of the ÷ operator. The quantity on the right of ÷ is 4×3,

i.e. 4 multiplied by whatever is on the right of ×. Since that is just 3, we see that we are dividing 12 by 4×3, i.e. by 12.

Note that monadic and dyadic operators can be intermixed in an expression. For example, 4+-30÷5+6>7×⊛8 is a complicated-looking expression. Let us add parentheses according to the right-associativity convention: 4+(-(30÷(5+(6>(7×(⊛8)))))). (We know that the minus sign is an operator since "minus thirty" would have been written ¯30. You cannot interchange ¯ and - since ¯30 has an effect only on the magnitude of "thirty," while -3÷5+6>7×8 changes the sign of <u>everything</u> to its right. We know that - and ⊛ are monadic operators since each is preceded by some other operator rather than an operand.)

Now to evaluate the expression. We start with the most nested subexpression. ⊛8 is the natural logarithm of 8, i.e., 2.0794415417 and 7×2.0794415417 ↔ 14.5560907919. Next, 6>14.5560907919 ↔ 0. 5+0 ↔ 5, and 30÷5 ↔ 6. Next, ¯6 ↔ -6 and 4+¯6 ↔ ¯2. So we see that 4+-30÷5+6>7×⊛8 ↔ ¯2.

Expressions are never evaluated backwards in APPLE, although they <u>are</u> evaluated from the right. When you write 4-5-6, the result is 5 since 5-6 ↔ ¯1 and 4-¯1 ↔ 5.

CHAPTER SIX

ELEMENTARY ARRAY MANIPULATIONS

In this chapter, you will be introduced to a class of operators that are useful in manipulating arrays. These manipulations include the familiar process of extracting one or more elements from an array by subscripting. Since those elements that have been extracted are arrays, recall that scalars are rank-0 arrays, you will be forming a subarray of the original array each time you subscript into it. The subarray may consist of more than one element.

Subscripting is only one means of forming subarrays of an original array. You will be introduced to techniques for forming subarrays consisting of elements satisfying some set of properties, as well as techniques for taking various cross sections of an array.

We will also describe ways of rotating and transposing arrays, of combining several arrays to make a bigger array, and of changing the dimensionality of arrays.

6.1 Index Origin

Subscripting is the process by which you specify one or more elements of an array. The subscript of a specific array element is known as the index of that element.

There is some confusion between the way programming languages refer to the first element in a vector. In some languages, that element has an index of 1, while its index is 0 in other languages. Depending on the programmer's particular needs, one of these indexing conventions is often preferable over the other.

Since only you, the programmer, know which indexing convention is preferable for your personal application, APPLE leaves the choice up to you. The index origin is the value of the index of the first elements of a vector. The index origin is contained in the rank-0 array *IORG*. *IORG* normally contains the value 1. If you want to specify its value, you begin your program with either

```
      IORG ← 0
   or
      IORG ← 1
```

The value of *IORG* will remain constant throughout the body of your program. For the time being, the values of *IORG* are restricted to 0 or 1. Eventually, the language may be extended to permit arbitrary integral values of *IORG*.

6.2 Indexing of Arrays

In the following subsections, you will be given the necessary vocabulary and notations for subscripting arrays.

6.2.1 The Empty Vector

Recall that if X is some array and $\rho X \leftrightarrow 0$, then X is an empty vector (since X contains no elements, and $\rho\rho X \leftrightarrow 1$, so X is a vector).

We will be using the empty vector often enough to require giving it a name for easy reference. The name of the empty vector is $\bar{\epsilon}$.

6.2.2 Vector Index Generation

Suppose you wanted to generate a vector that consists of all of the permissible values of indices, in ascending order, for some vector V. Clearly, such a vector consists of ρV elements. If *IORG* $\leftrightarrow 0$, then this vector is $(0,1,\ldots,\bar{}1+\rho V)$ if *IORG* $\leftrightarrow 1$, then this vector is $(1,2,\ldots,\rho V)$.

In APPLE, you need only write $\iota\rho V$ to produce this vector. The operator ι produces the desired vector. In fact, all you need to do to get a vector of length n, where n≥0, is write ιn.

Example: If *IORG* ↔ 1, then

```
⍳5 ↔ (1,2,3,4,5)
⍳1 ↔ (1)
```

while if *IORG* ↔ 0 then

```
⍳5 ↔ (0,1,2,3,4)
⍳0 ↔ (0)
```

If we write ⍳0, from the definition of ⍳, we should get a vector of length 0. A vector of length 0 can only be the empty vector $\bar{\epsilon}$. Hence, regardless of whether *IORG* ↔ 1 or *IORG* ↔ 0, we will always have ⍳0 ↔ $\bar{\epsilon}$.

So far, ⍳*N* is defined only for nonnegative integers *N*, where *N* is either a scalar or a one-element vector. We will soon extend the definition of ⍳ to cover all rank-0 and rank-1 arrays consisting of nonnegative integers.

6.2.3 Subscripting of Vectors

If you want to indicate the k-th element of some vector V, where $1 \leq k \leq \rho V$, the appropriate index would be $k+IORG-1$. For example, if you want the first element and $IORG \leftrightarrow 1$, you want the index to be 1+1-1, i.e. 1; while if $IORG \leftrightarrow 0$, then you want the index to be 1+0-1, i.e. 0. Correspondingly, if you wanted the fifth element, the index would be either 5 or 4. You should convince yourself that $k+IORG-1$ is always an element of $\iota\rho V$.

Scalar subscripting is exactly like subscripting in other programming languages. In order to select the k-th element of V, you write $V(K)$, where K is the appropriate index corresponding to $IORG$ and k. For example, suppose $IORG \leftrightarrow 1$ and $V \leftrightarrow (1,5,7,9,3,4,1)$. Then $V[1] \leftrightarrow 1$, $V[3] \leftrightarrow 7$, $V[6] \leftrightarrow 4$. If $IORG \leftrightarrow 0$, then $V[1] \leftrightarrow 5$, $V[3] \leftrightarrow 9$, and $V[6] \leftrightarrow 2$.

When you subscript a vector with a scalar, the result is a scalar.

You are not restricted to using scalars as subscripts, however. If you subscript a vector with a vector, the result is a vector of the same dimensionality as the subscript vector. For example, if $A \leftrightarrow [2,1,5,4]$ and V is the vector we used in the previous example, if $IORG \leftrightarrow 1$, $V[A] \leftrightarrow (5,1,3,9)$ and $IORG \leftrightarrow 0$, then $V[A] \leftrightarrow (7,5,4,3)$.

When you subscript a vector with an array, the result has the same dimensionality as the subscripting array.

Example:

If $\underline{IORG} \leftrightarrow 1$ and W is the rank-2 array

$$W \leftrightarrow \begin{matrix} 7 & 3 & 5 & 5 \\ 2 & 1 & 1 & 2 \end{matrix}$$

then

$$V[W] \leftrightarrow \begin{matrix} 2 & 7 & 3 & 3 \\ 5 & 1 & 1 & 5 \end{matrix}$$

6.2.4 Subscripting of Arrays

The number of elements in an array A is the product of the elements of the vector ρA. Since A has $\rho\rho A$ coordinates, any subscript of A must be composed of $\rho\rho A$ components. These are separated from one another by the delimiter (;). The first subscript you list applies to the first coordinate of A; the second one applies to the second coordinate, and so forth. If $K \leftrightarrow \underline{IORG}+k-1$ and I the k-th coordinate of A, it is required that I be in the range $\underline{IORG} \leq I \leq \underline{IORG} + (\rho A)[K]-1$. That is, a coordinate subscript must lie in the range $1,\ldots,(\rho A)[K]$ if $\underline{IORG} \leftrightarrow 1$, or $0,\ldots,(\rho A[K]-1$ if $\underline{IORG} \leftrightarrow 0$. This is always equivalent to saying that a coordinate subscript is an element of the vector $\iota\rho A[K]$.

Example:

If *IOREG* ↔ 1 and *A* is the rank-2 array

```
A ↔ 3 4 5 6
    8 7 2 1
```

Then ρ*A* ↔ (2, 4)

The subscript for the first coordinate must be either 1 or 2; the subscript for the second coordinate must be 1, 2, 3 or 4. When you write *A*[1;1], you specify the scalar 3. We also have *A*[2;3] ↔ 2, *A*[2;4] ↔ 6 and *A*[2;1] ↔ 8.

Just as you can subscript a vector with an array, you can also subscript an array with an array. The result is an array *B* whose rank ρρ*B* is equal to the sum of the ranks of the coordinate subscripts. The dimensionality of *B* , ρ*B* , is the vector that is composed of the dimension vector for the first coordinate subscript, followed by the dimension vector for the second coordinate subscript, and so forth.

For example, we can subscript *A* with the vector (1, 2) for the first coordinate and the scalar 3 for the second coordinate, thereby producing the array composed of *A*[1;3] and *A*[2;3]. The result must be a rank-1 array since ρρ(1,2) ↔ 1 and ρρ3 ↔ 0 and 1 + 0 ↔ 1. The dimension vector of the result is the vector (2, $\bar{\epsilon}$) ↔ (2) since ρ(1, 2) ↔ 2and ρ3 ↔ $\bar{\epsilon}$. Therefore, we must have *A*[1 2; 3] ↔ (5, 2)

APPLE permits you to write vectors in subscript expressions

either with or without parentheses. Of course, for the sake of clarity, you could have written $A[(1, 2);3]$ if you had wished.

Now, suppose we have a matrix

$$C \leftrightarrow \begin{matrix} 1 & 1 & 2 \\ 2 & 1 & 2 \end{matrix}$$

so that $\rho C \leftrightarrow (2, 3)$. When we write $A[C;3\ 4\ ]$, what should the result, D, be?

We know that $\rho\rho D \leftrightarrow 3$ (since $\rho\rho B \leftrightarrow 2$ and $\rho\rho(3,4) \leftrightarrow 1$), and $\rho D \leftrightarrow (2,3,2)$ the catenation (composition) of ρC and $\rho(3,4)$). For simplicity, call $E \leftrightarrow (3,4)$. We can deduce the elements of D, as follows. $D[1;1;1]$ must correspond to the $C[1;1]$-th row of A and the $E[1]$-th column of A. Hence,

$$D[1;1;1] \leftrightarrow A[1;3] \leftrightarrow 5$$

Similarly, we obtain:

$$\begin{aligned}
D[1;1;2] &\leftrightarrow A[1;4] \leftrightarrow 6 \\
D[1;2;1] &\leftrightarrow A[1;3] \leftrightarrow 5 \\
D[1;2;2] &\leftrightarrow A[1;4] \leftrightarrow 6 \\
D[1;3;1] &\leftrightarrow A[2;3] \leftrightarrow 2 \\
D[1;3;2] &\leftrightarrow A[2;4] \leftrightarrow 1 \\
D[2;1;1] &\leftrightarrow A[2;3] \leftrightarrow 2 \\
D[2;1;2] &\leftrightarrow A[2;4] \leftrightarrow 1 \\
D[2;2;1] &\leftrightarrow A[1;3] \leftrightarrow 5 \\
D[2;2;2] &\leftrightarrow A[1;4] \leftrightarrow 6 \\
D[2;3;1] &\leftrightarrow A[2;3] \leftrightarrow 2 \\
D[2;3;2] &\leftrightarrow A[2;4] \leftrightarrow 1
\end{aligned}$$

We can graphically represent D as

```
        5 6
        5 6
        2 1
D ↔
        2 1
        5 6
        2 1
```

Finally, suppose you want the first plane of D. You could write $D[1;1\ 2\ 3;\ 1\ 2]$. The result would be a rank-2 array of dimensionality $(3,2)$. (Why?) It would consist of $D[1;1;1]$, $D[1;1;2]$ and so forth, as expected. Recalling that $\iota 3 \leftrightarrow (1,2,3)$ and $\iota 2 \leftrightarrow (1,2)$, you could also write $D[1;\iota 3;\iota 2]$.

> By convention, instead of writing $\iota(\rho A)[K]$ as a subscript for the k-th coordinate of an array A, you can elide* the subscript for that coordinate, writing any required semicolon separators as you normally would.

Formally, we have the equivalences

$$A[;J;K;\ldots;Y;Z] \leftrightarrow A[\iota(\rho A)[\underline{IORG}];J;K\ldots Y;Z]$$
$$A[I;;K;\ldots Y;Z] \leftrightarrow A[I;\iota(\rho A)[1+\underline{IORG}];K;\ldots Y;Z]$$
$$\vdots$$
$$A[I;J;K;\ldots;Y;] \leftrightarrow A[I;J;K;\ldots;Y;\iota(\rho A)[\underline{IORG}+\bar{}1+\rho\rho A]]$$

Hence, instead of writing $D[1;\iota 3;\iota 2]$, you can write $D[1;;]$, and the APPLE compiler will deduce the content of the elided coordinate subscripts.

---

* Elide--to omit.

6.3 The Ravel Operator

When we constructed the array D in Section 6.2.4, the order in which we listed its elements was significant. We started with the element whose subscript was *IORG* in each coordinate. Subsequently, we allowed the right-most coordinate subscript to vary most rapidly, then the subscripts in the coordinate field second from the right, and so on. This ordering is called an odometer ordering since the indices appear in the order they would follow had they been placed on the individual wheels of an automobile odometer. (Wheel K of the odometer is numbered with the elements of $\iota(\rho D)[K]$, starting with *IORG*.)

It is sometimes useful to view an array as a vector. In APPLE, the comma (,)is used monadically to represent the ravel operator. The ravel of an array is the vector whose elements are those of the original array in the odometer order. In particular, the ravel of a scalar is the vector whose only element is the scalar.

Thus, we see that

$$,D \leftrightarrow (5,6,5,6,2,1,2,1,5,6,2,1)$$

6.4 Array Index Generation

Here, we generalize upon the definition of the monadic operator ι. Suppose A is some array and $N \leftrightarrow \rho A$, so that N is a vector.

Then, we define $\underline{\iota}A$ to be the matrix of dimensionality $\rho(\underline{\iota}N) \leftrightarrow ((\rho,A),(\rho\rho A))$. That is $\underline{\iota}N$ has as many rows as there are elements in A and as many columns as the rank of A. Thus, $\underline{\iota}N$ contains a column for every component of a subscript of A and a row for every element of A.

The rows of $\underline{\iota}N$ are in odometer order, so that the i-th row of $\underline{\iota}N$ is the index associated with the element of A that corresponds to the i-th element of $,A$ (i.e., $(\underline{\iota}N)[I]$ is the index in A of $(,A)[I]$).

Example: Let us refer back to the array D constructed in Section 6.2.4. $\rho D \leftrightarrow (2,3,2)$, so let us look at $\underline{\iota}\rho D$.

We have

$\underline{\iota}\rho D \leftrightarrow$

```
1 1 1
1 1 2
1 2 1
1 2 2
1 3 1
1 3 2
2 1 1
2 1 2
2 2 1
2 2 2
2 3 1
2 3 2
```

Now let us look at a few examples of the concordance between $\underline{\iota}\rho D$ and D.

First,

$D \leftrightarrow$

```
5 6
5 6
2 1

2 1
5 6
2 1
```

and $,D \leftrightarrow (5,6,5,6,2,1,2,1,5,6,2,1)$. Now, $(,D)[3] \leftrightarrow 5$ and $(\iota\rho D)[3;] \leftrightarrow (1,2,1)$. We see that $D[1;2;1] \leftrightarrow 5$.

We also have, $(,D)[11] \leftrightarrow 2$, $(\underline{1}\rho D)[11;] \leftrightarrow (2,3,1)$ and $D[2;3;1) \leftrightarrow 2$.

In this example, we assumed that $\underline{IORG} \leftrightarrow 1$. If $\underline{IORG} \leftrightarrow 0$, it follows that

$\underline{1}\rho D \leftrightarrow$

```
0 0 0
0 0 1
0 1 0
0 1 1
0 2 0
0 2 1

1 0 0
1 0 1
1 1 1
1 1 2
1 2 0
1 2 1
```

6.5 The Subscript Generator

Suppose A is an array and I is some row of $\underline{1}\rho A$. There is an element of I for each component of A. But in order to subscript A by I, there must be semicolon delimiters present between the elements of I.

The monadic operator ;/ is used to generate scalar subscripts. Its only action is placing a semicolon between the elements of a vector.

Thus, if $\rho A \leftrightarrow (4,7,3,5,8)$, the vector $I \leftrightarrow (2,3,2,1,6)$ is certainly a row of the matrix $\underline{1}\rho A$. We have

$A[;/I] \leftrightarrow A[2;3;2;1;6]$.

6.6 Partial Subscripting

Suppose *A* is an array and *K* is an element of ⍳⍴⍴*A*, that is, *K* is an element of (0,1,...,¯1+⍴⍴*A*) if *IORG* ↔ 1, or an element of (0,1,...,¯1⍴⍴*A*) if *IORG* ↔ 0. Suppose that *I* is some array in which each element of *I* is an element of ⍳(⍴*A*)[*K*] so that *I* is a valid subscript array for component *K* of *A*. Then, if the only component of *A* that you want to subscript is component *K*, you may do so by writing *A*[[*K*]*I*].

K must be an integer scalar, or the integer scalar content of a variable.

Example: Suppose *IORG* ↔ 1 and *A* is the rank-3 array.

```
          3 4 5
          2 1 3
          0 7 6
         -2 4 8
A ↔
          5 9 2
          1 1 3
          4 0 7
          3 9 8
```

Then ⍴*A* ↔ (2,4,3). Now, [1;;] is the first plane of *A*. This could be written [[1]1], which says the subscript 1 is to be applied to coordinate [1] of *A* only.

When we write *A*[[2]2 3], have ⍴*A*[[2]2 3] ↔ (2,2,3)

```
                2 1 3
                0 7 6
A[[2]2 3] ↔
                1 1 3
                4 0 7
```

We can also look at *A*[[3]2 3].

Here, ρA[[3]2 3] ↔ (3, 4, 2) and

```
                4 5
                1 3
                7 6
A[[3]2 3] ↔
                9 2
                1 3
                0 7
                9 8
```

6.7 Reshaping Arrays

Any array can be transformed into a vector by use of the ravel operator. Any array A can also be transformed into an array B of different dimensionality. To do this you use the reshape operator ρ. The reshape operator is dyadic, while the shape operator ρ that yields the dimension vector of an array--its shape--is a monadic operator.

When you want to transform A into B so that $\rho B \leftrightarrow R$, where R is some vector of nonnegative integers, you write $B \leftarrow R\rho A$

B must contain a number N of elements equal to the product of the elements of R since $\rho B \leftrightarrow R$. B receives its elements from A according to the formula:

$$B[;/(\underline{\iota}\rho B)[I]] \leftrightarrow A[;/(\underline{\iota}\rho A)[\underline{IORG}+N|I-\underline{IORG}]]$$

This formula is another way of saying that a vector V is constructed from enough repetitions and partial repetitions of the elements of $,A$ that V contains as many elements as B will contain. In odometer order, the first element of B will be the

first element of V, the second element of B will be the second element of V, and so forth.

Example: Suppose $\underline{IORG} \leftrightarrow 1$ and A is the rank-3 array

$$A \leftrightarrow \begin{array}{cccc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 \end{array}$$

Then $\rho A \leftrightarrow (2,3,4)$.

Recall that

$\iota 24 \leftrightarrow (1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24)$

If $\iota 24$ were reshaped so that its dimension were $(2,3,4)$, it would be equivalent to A. That is,

$(2,3,4)\rho\iota 24 \leftrightarrow A$.

The vector $\iota 60$ has 60 elements. But, when we write $(2,3,4)\rho\iota 60$, we are only using the first $2\times 3\times 4 \leftrightarrow 24$ elements of $\iota 60$. Therefore,

$(2,3,4)\rho\iota 60 \leftrightarrow (2,3,4)\rho\iota 24 \leftrightarrow A$.

Verify that

$$(2,3,4)\rho\iota 7 \leftrightarrow \begin{array}{cccc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 1 \\ 2 & 3 & 4 & 5 \\ 6 & 7 & 1 & 2 \\ 3 & 4 & 5 & 6 \\ 7 & 1 & 2 & 3 \end{array}$$

Note also that

```
                 1  2  3  4
                 5  6  7  8
                 9 10 11 12
(6,4)ρA ↔
                12 14 15 16
                17 18 19 20
                21 22 23 24

                 1  2  3   4  5  6
                 7  8  9  10 11 12
(4,6)ρA ↔
                12 14 15 16 17 18
                19 20 21 22 23 24
```

Finally, $(\iota 0)\rho 3$ is the scalar 3. This is true since

$\rho(\iota 0)3 \leftrightarrow \bar{\epsilon}$ and $\rho\rho(\iota 0)\rho 3 \leftrightarrow \rho\bar{\epsilon} \leftrightarrow 0$

6.8 Arithmetic Array Manipulations

This section introduces the very useful classes of operators that enable you to perform arithmetic processes on the elements of an array.

6.8.1 Vector Reduction

It is often desirable to obtain the sum or product of the elements of a vector. In APPLE, you write $+/V$ in order to obtain the sum of the elements of V ; you write $\times/V$ in order to obtain the product of the elements of V .

The sum of the elements of a vector is usually considered to be a scalar. In APPLE, while V is a vector, $+/V$ is a scalar. Thus, the rank of $+/V$ is equal to the rank of V reduced by 1. For this reason, $+/V$ is read as the "plus reduction of V."

Example:

```
+/(2,3,4,7,¯5,2) ↔ 13
×/(2,3,4,7,¯5,2) ↔ ¯1680
```

You can use any of the operators from Table III (see Section 5.3) in conjunction with the slash to form a reduction operator. The effect of reduction is to place the dyadic scalar operator between the elements of the vector operand, and then to evaluate the resulting expression. Consequently, -/(2,3,4,7,¯5,2)↔2-3-4-7-¯5-2↔¯11. If $\rho V \leftrightarrow 1$ for any operator $*$, $*/V \leftrightarrow (\iota 0)\rho V$. If $\rho V \leftrightarrow 0$, for any operator $*$, $*/V \leftrightarrow (\iota 0)\rho I$, where I is the identity element associated with $*$.

It is useful to note that $\times/\rho A$ is the number of elements in the array A. When A is a scalar, $\rho A \leftrightarrow \bar{\epsilon}$. Since $\times/\rho A$ is always equal to the number of elements in any array, regardless of its rank.

Note that we always have $\lfloor/\iota N \leftrightarrow \underline{IORG}$ for any N, and $\lceil/\iota N \leftrightarrow N-1+\underline{IORG}$. We will use this notation frequently when we write subscript expressions that are independent of the value of $\underline{IORG}$.

6.8.2 Array Reduction

Suppose A is a rank-n array and suppose $*$ is a scalar dyadic operator from Table III (see Section 5.3). Then $*/[K]A$ is the application of $*$ over the elements of coordinate K of A. The

result is an array of rank N-1. Its dimension vector is obtained from ρA by suppressing $(\rho A)[K]$.

Example: Suppose *IORG* ↔ 1 and ρA ↔ (2,4) where

```
A ↔ 2 5 7 10
    4 3 1  2
```

Then

```
+/[1]A ↔ (6,8,8,12)
+/[2]A ↔ (24,10)
-/[1]A ↔ (¯2,2,6,8)
-/[2]A ↔ (¯6,0)
```

Suppose ρB ↔ (3,3,3) where

```
          1  2  3
          4  5  6
          7  8  9

         10 11 12
B ↔      13 14 15
         16 17 18

         19 20 21
         22 23 24
         25 26 27
```

Then

```
                30 33 36
+/[1] B ←→      39 42 45
                48 51 54

                12 15 18
+/[2] B ←→      39 42 45
                66 69 72

                 6 15 24
+/[3] B ←→      33 42 51
                60 69 78

                19 20 21
⌈/[1] B ←→      22 12 14
                25 26 27

                 7  8  9
⌈/[2] B ←→      16 17 18
                25 26 27

                 3  6  9
⌈/[3] B ←→      12 15 18
                21 24 27

                 1  2  3
⌊/[1] B ←→       4  5  6
                 7  8  9

                 1  2  3
⌊/[2] B ←→      10 11 12
                19 20 21

                 1  4  7
⌊/[3] B ←→      10 13 16
                19 22 25
```

When K is the last coordinate of A, you may elide the [⌈/⍳⍴⍴A] and write ⌈/A. Hence, if A is the matrix from the example, we

have

$$+/[2]A \leftrightarrow +/A$$

If A is any one-element array (i.e., $\times/\rho A \leftrightarrow 1$, then $\divideontimes/[K]A$ is a one-element array of rank $0\lceil\bar{}1+\rho\rho A$. For example $\div/(1,1,1,1,1)\rho 3 \leftrightarrow (1,1,1,1)\rho 3$ while $\div/(\iota 0)\rho 3 \leftrightarrow (\iota 0)\rho 3$.

6.8.3 Vector Accumulation

The accumulation operator \ is the analogue of the reduction operator. If $\divideontimes$ is one of the operators from Table III (see Section 5.3), $\divideontimes\backslash V$ is the $\divideontimes$-reduction of the elements of V, starting with the i-th element of V. In other words, $\divideontimes\backslash V$ is a vector of the "partial sums" obtained in the evaluation of $\divideontimes/V$.

Example: Suppose

$$V \leftrightarrow (5,4,3,2,1)$$

then

$$\times\backslash V \leftrightarrow (120,\ 24,\ 6,\ 2,\ 1)$$

and

$$-\backslash V \leftrightarrow (3,\ 2,\ 2,\ 1,\ 1)$$

6.8.4 Array Accumulation

As in the case of array reduction, $\divideontimes\backslash[K]A$ is the application of $\divideontimes$ over the elements of coordinate K of A and $\rho\divideontimes\backslash[K]A \leftrightarrow A$.

Example: If *IORG* $\leftrightarrow$ 1 and M is the rank-2 array

```
                2 _1 _77 _3
M ↔→            9 _1 _10 _2
```

Then

```
                11  0  _3  1
+\[1]M ↔→       9 _1 _10 _2

                13 _11 _10 _3
+\[2]M ↔→       4 _13  _12 _2


                 9  1  _7 _3
⌈\[1]M ↔→       9  1 _10 _2

                 7  7  _7 _3
⌈\[2]M ↔→       9 _2   _2 _2
```

6.9 The Catenation of Vectors

If V and W are any two vectors, then you can construct a new vector $X \leftrightarrow V,W$ where $\rho V,W \leftrightarrow (\rho V)+\rho W$ and $X[\iota\rho V] \leftrightarrow V$

$$X[(\rho W)\rho(\rho V)+\iota\rho W] \leftrightarrow W$$

That is, the first ρV elements of V,W are the elements of V and the last ρW elements of V,W are the elements of W.

V,W is called the <u>catenation</u> of V and W. The catenation operator(,) is a dyadic operator. If either V or W is a scalar, it is treated as a vector, so that V,W is a vector of dimension $(\rho,V)+\rho,W$.

Example: If U is the scalar 1 and V and W are the vectors

$$V \leftrightarrow (2,3,4)$$
$$W \leftrightarrow (5,6,7,8)$$

Then

```
      V,W ↔ (2,3,4,5,6,7,8)
      U,W ↔ (1,5,6,7,8)
    U,V,W ↔ (1,2,3,4,5,6,7,8)
```

6.10 The Interval Operator (Optional on First Reading)

The symbol ⍷ is used to represent the monadic operator that produces an interval vector. The argument of ⍷ is a vector *LEN,ORG,S*. ⍷ *LEN,ORG,S* is the vector of length *LEN* whose least element equals *ORG*. *S* must be either 0 or 1. If *S* ↔ 0, successive elements increase by 1; if otherwise, the elements decrease by 1.

Note that ⍷ is independent of the value of *IORG*.

```
⍷ 5,3,0 ↔ (3,4,5,6,7)
⍷ 5,3,1 ↔ (7,6,5,4,3)
```

6.11 The Subarray Function (Optional on First Reading)

It is often desirable to work with a subarray *B* of an array *A* where ρρ*B* ↔ ρρ*A* and the hyperplanes of *B* are adjacent hyperplanes of *A*. For example, if ρ*A* ↔ (3,5,7) we might want to construct an array

B ↔ *A*[1 2;4 3 2;3 4 5] ↔ *A*[⍷ 2,1,0;⍷ 3,2,1;⍷ 3,3,3]

The dyadic operator ∆ is useful for this purpose. The right argument of ∆ is *A* and the left argument is a rank-2 array *F* where ρ*F* ↔ ((ρρ*A*),3). The rows of *F* are the vectors required by the interval operator ⍷ in the expression above. That is, the elements of *F* satisfy the formula

$$B \leftrightarrow F\Delta A \leftrightarrow A[⍤\ F[\underline{IORG};];⍤\ F[\underline{IORG}+1;];...;⍤\ F[\lceil/\iota\rho\rho A;]]$$

Thus, for the example, F would be the matrix

$$F \leftrightarrow \begin{matrix} 2 & 1 & 0 \\ 3 & 2 & 1 \\ 3 & 3 & 3 \end{matrix}$$

6.11.1 The Whole Array Operator (Optional)

The monadic use of Δ on an array A produces the matrix F such that $F\Delta A \leftrightarrow A$. F satisfies the following conditions:

$$\begin{aligned} \rho F &\leftrightarrow (\rho\rho A),3 \\ F[;\lfloor/\iota 3] &\leftrightarrow (\rho A) \\ F[;1+\lfloor/\iota 3] &\leftrightarrow (\rho\rho A)\rho\underline{IORG} \\ F[;\lceil/\iota 3] &\leftrightarrow (\rho\rho A)\rho 0 \end{aligned}$$

Example:

If $\rho A \leftrightarrow (2,3,5,7)$ and $\underline{IORG} \leftrightarrow 1$
Then

$$\Delta A \leftrightarrow \begin{matrix} 2 & 1 & 0 \\ 3 & 1 & 0 \\ 5 & 1 & 0 \\ 7 & 1 & 0 \end{matrix}$$

If $\rho B \leftrightarrow (3,5,6,8,2)$ and $\underline{IORG} \leftrightarrow 0$
Then

$$\Delta B \leftrightarrow \begin{matrix} 3 & 0 & 0 \\ 5 & 0 & 0 \\ 6 & 0 & 0 \\ 8 & 0 & 0 \\ 2 & 0 & 0 \end{matrix}$$

6.11.2 The Cross Section Operator (Optional)

An array cross section is obtained when all of the component subscripts are either scalars or elided. For example, if *A* is a rank-4 array, the following are some possible cross sections of *A*.

```
A
A[1;;;]
A[;1;3;]
A[1;;2]
A[1;2;3;1]
```

The dyadic cross section operator ⍙ is primarily used for formalizing the subscripting of an array by scalars. The right argument of ⍙ is the array to be subscripted. The left argument of ⍙ is a rank-2 array *G*, $\rho G \leftrightarrow (\rho\rho A),2$. The elements of the first column of *G* are either 0 or 1 as follows:

If coordinate *K* is to be elided, then $G[K;] \leftrightarrow (0,0)$

If coordinate *K* is to be subscripted by the scalar *S*, then

$$G[K;] \leftrightarrow (1,S)$$

Example: If $\rho A \leftrightarrow (2,3,5,7)$, *IORG* $\leftrightarrow$ 1, and

$$G \leftrightarrow \begin{matrix} 1 & 2 \\ 0 & 0 \\ 1 & 4 \\ 1 & 6 \end{matrix}$$

Then

$$G \underline{\Delta} A \leftrightarrow A[2;;4;6]$$

6.12 Compression and Expansion

6.12.1 The Logical Compression of a Vector

Suppose U is a vector whose elements belong to the set {0,1}. Then U is called a logical vector. Then if X is any vector such that $\rho U \leftrightarrow \rho X$, we can form the subvector U/X (read "the U compression of X"), where $\rho U/X \leftrightarrow +/U$. The elements of U/X are the $X[I]$ such that $U[I] \leftrightarrow 1$.

Example: If U and V are the vectors

$$V \leftrightarrow (1,2,3,5,7,11,13)$$
$$U \leftrightarrow (1,0,0,1,0,1,1)$$

then

$$U/V \leftrightarrow (1,5,11,13)$$

since

$$\sim U \leftrightarrow (0,1,1,0,1,0,0)$$
$$(\sim U)/V \leftrightarrow (2,3,7)$$

6.12.2 The Logical Compression of an Array

The logical compression operator is extended to arrays, as follows. Let A be an array, and let U be a logical vector such that for some component I of A, $\rho U \leftrightarrow (\rho A)[I]$. Then, $U/[I]\ A$ the U compression along coordinate I of A is defined as

$$U/[I]\ A \leftrightarrow A[[I]\ U/\iota(\rho A)[I]]$$

Example: Suppose *IORG* ↔ 1 and *A* and *U* are

```
      U ↔ (1,1,0)

            1  2  3
            4  5  6
            7  8  9

           10 11 12
      A ↔  13 14 15
           16 17 18

           19 20 21
           22 23 24
           25 26 27
```

Then

```
                1  2  3
                4  5  6
                7  8  9
U/[1] A ↔
               10 11 12
               13 14 15
               16 17 18
```

and

```
                1  2
                4  5
                7  8

               10 11
U/[2] A ↔      13 14
               16 17

               19 20
               22 23
               25 26
```

and

```
                1  2  3
                4  5  6

U/[3] A ↔      10 11 12
               13 14 15

               19 20 21
               22 23 24
```

If $I \leftrightarrow \lceil/\iota\rho\rho A$ then $U/[I]\ A$ may be written as U/A.

6.12.3 The Logical Expansion of a Vector

The logical expansion of a vector is the analogue of the logical compression of the vector. In this case, if V is any vector and U is a logical vector, where $\rho V \leftrightarrow +/U$, then U/V is the vector having zeros wherever U has zeros, and whose remaining elements are taken in order from V.

Example:

$$(1,1,0,1,0)\backslash(1,2,4) \leftrightarrow (1,2,0,4,0)$$

6.12.4 The Logical Expansion of an Array

The logical expansion operator is extended to arrays, as follows. Let A be an array and let U be a logical vector such that for some component I of A, $+/U \leftrightarrow (\rho A)[I]$. Then $U\backslash[I]A$, the U expansion along coordinate I of A, is defined as the array whose dimension is given by

$$\rho(U\backslash[I]A)[I] \leftrightarrow \rho U$$

and for K an element of $\iota\rho\rho A$, $\rho(U\backslash[I]A)[K] \leftrightarrow (\rho A)[K]$

where $K \neq I$ and for every element J of $\iota\rho U$,

$$(U\backslash A)[[I]J] \leftrightarrow A[[I]+/U[\iota J]]\times U[J]$$

The following example clarifies the situation.

Example: Suppose *IORG* ↔ 1 and *U* and *A* are

```
U ↔ (1,0,1,0)

      2 4
      6 8
A ↔
      1 3
      5 7
```

Then

```
      2 4
      6 8

      0 0
      0 0

U\[1]A ↔
      1 3
      5 7

      0 0
      0 0
```

and

```
      2 4
      0 0
      6 8
      0 0

U\[2]A ↔
      1 3
      0 0
      5 7
      0 0

      2 0 4 0
      6 0 8 0
U\[3]A ↔
      1 0 3 0
      5 0 7 0
```

If *I* ↔ ⌈/ιρρ*A*, then *U*\[*I*]*A* may be written as *U**A*.

6.12.5 The Relation Between Expansion and Compression

The following relation occurs between the expansion and compression of any array A by any logical vector U.

$$U/[I]U\backslash[I]A \leftrightarrow A$$

6.13 Prefix and Suffix Vectors

A prefix vector is a logical vector P whose first R components are ones and whose remaining $(\rho P)-R$ components are zeros. A suffix vector is a logical vector S whose last R components are ones and whose remaining $(\rho S)-R$ components are zeros.

The dyadic operator for specifying a prefix vector is α; the one for specifying suffix vectors is ω. In both cases, the left argument L is the length of the vector to be constructed, and the right argument R is the number of ones required in the vector.

Example:

```
7α3 ↔ (1,1,1,0,0,0,0)
7ω3 ↔ (0,0,0,0,1,1,1)
```

Prefix vectors are useful for selecting the first R components along some coordinate of an array; suffix vectors are useful for selecting the last R components along some coordinate of an array.

Example: Suppose *IORG* ↔ 1 and *A* is the array

```
      1  2  3  4  5
      6  7  8  9 10
A ↔  11 12 13 14 15
     16 17 18 19 20
```

Then

```
(4⍺2)/[1]A ↔  1  2  3  4  5
              6  7  8  9 10

(4⍵2)/[1]A ↔ 11 12 13 14
             15 16 17 18

              3  4  5
              8  9 10
(5⍵3)/[2]A ↔ 13 14 15
             18 19 20
```

6.14 The Monadic Transpose Operator

If *A* is any rank-n array, where n≥2, the monadic transpose of *A*, written ⍉*A*, is equivalent to *A* with its last two coordinates interchanged. If *A* is a matrix, then ⍉*A* is the transpose of *A*.

Formally, for (⍴⍴*A*)≥2,

$$\rho\,⍉A \leftrightarrow \rho A[\iota\,{}^{-}2+\rho\rho A),\ (\lceil/\iota\rho\rho A),\ {}^{-}1+\lceil/\iota\rho\rho A)$$

and for any row *L* of ⍳⍴⍉*A*, we have

$$A[;/L] \leftrightarrow (⍉A)[;/L[M]]$$

where

$$M \leftrightarrow (\iota\,{}^{-}2+\rho\rho A),\ (\lceil/\iota\rho\rho A),\ {}^{-}1+\lceil/\iota\rho\rho A)$$

Example: If

```
      1  2  3  4
      5  6  7  8
A ↔
      9 10 11 12
     13 14 15 16
```

Then

```
      1  5
      2  6
      3  7
      4  8
⍉A ←→
      9 13
     10 14
     11 15
     12 16
```

6.15 The Take Operator

The take operator is a generalization of the prefix and suffix operators. It permits you to form a subarray B of an array A by writing $T\uparrow A$, where T is a vector of integers such that $\rho T \leftrightarrow \rho\rho A$ and $(|T) \leq \rho A$.

The effect of the take operator is that

$$\rho B \leftrightarrow |\rho A$$

and coordinate I of B receives the first $T[I]$ elements from coordinate I of A if $T[I]\geq 0$; otherwise B receives the last $T[I]$ elements of coordinate I of A when $T[I]<0$.

The take operator can be expressed, as follows.

$$T\uparrow A \leftrightarrow F\Delta A$$

where

$$F \leftrightarrow ⍉(3,\rho\rho A)\rho(|T),(((\rho T)\rho IORG)+(T<0)\times(\rho A)-|T),(\rho T)\rho 0$$

APPLE has a convention that makes it easier to write certain take vectors. If you only want to subset the last few

coordinates of A, you may write $T\uparrow A$ where $(\rho T)<\rho\rho A$. This is equivalent to writing $(((\sim(\rho\rho A)\alpha\rho T)/\rho A),T)\uparrow A$.

Also, if you want to subset a few coordinates of A and keep the rest intact, you may write $T\uparrow[I]A$ where I is the vector of coordinates in which you are interested. This is equivalent to writing $B\uparrow A$, where $\rho B \leftrightarrow \rho\rho A$ and $B[I] \leftrightarrow T$, while $(I\neq\iota\rho\rho A)/B \leftrightarrow (I\neq\iota\rho\rho A)/\rho A$.

Examples: Suppose $\underline{IORG} \leftrightarrow 1$ and $\rho\rho A \leftrightarrow 3$, where

$$A \leftrightarrow \begin{matrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 \end{matrix}$$

Then

$$(1,2,3)\uparrow A \leftrightarrow \begin{matrix} 1 & 2 & 3 \\ 5 & 6 & 7 \end{matrix}$$

$$(1,2,{}^{-}3)\uparrow A \leftrightarrow \begin{matrix} 2 & 3 & 4 \\ 6 & 7 & 8 \end{matrix}$$

$$({}^{-}1,{}^{-}2,{}^{-}2)\uparrow A \leftrightarrow \begin{matrix} 19 & 20 \\ 23 & 24 \end{matrix}$$

$$(2,{}^{-}3)\uparrow A \leftrightarrow \begin{matrix} 6 & 7 & 8 \\ 10 & 11 & 12 \\ 18 & 19 & 20 \\ 22 & 23 & 24 \end{matrix}$$

since, by convention, $(2,{}^{-}3)\uparrow A \leftrightarrow (2,2,{}^{-}3)\uparrow A$

Finally,

```
(1,¯2)↑[1,2]A ↔    5  6  7  8
                   9 10 11 12

                   5  6  7  8
                   9 10 11 12
¯2↑[2]A ↔
                  17 18 19 20
                  21 22 23 24
```

6.16 The Drop Operator

The drop operator ↓ is the analogue of the take operator. When you write $T{\downarrow}A$, the first or last $|T[I]$ components of coordinate I of A are suppressed according as $T[I]$ is positive or negative.

Formally,

$$T{\downarrow}A \leftrightarrow G\Delta A$$

where

$$G \leftrightarrow \Phi(3,\rho\rho A)\rho((\rho A)-|T,(((\rho T)\rho IORG)+0\lfloor T),(\rho\rho A)\rho 0$$

The conventions mentioned for the take operator also apply to the drop operator.

Examples: Suppose $IORG \leftrightarrow 1$ and A is the array defined in Section 6.15.

Then

```
(1,2,3)↓A ↔ (1,1,1)ρ3

                3  4
                7  8
               11 12
      2↓A ↔
               15 16
               19 20
               23 24
```

6.17 The Reversal Operator

If A is any array, then $\phi[I]A$ is the reversal of coordinate I of A. Formally,

$$\phi[I]A \leftrightarrow H\Delta A$$

where

$$H \leftrightarrow \Phi(3,\rho\rho A)\rho(\Delta A)[;\underline{IORG}],(\Delta A)[;1+\underline{IORG}],((\rho\rho A)\rho I)=\iota\rho\rho A$$

If $I \leftrightarrow \lceil/\iota\rho\rho A$, then I may be elided.

Example: As in Section 6.15, suppose $\underline{IORG} \leftrightarrow 1$ and

$$A \leftrightarrow (2,3,4)\rho\iota 24$$

Then

$\phi[1]A \leftrightarrow$

```
13 14 15 16
17 18 19 20
21 22 23 24

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

$\phi[2]A \leftrightarrow$

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

21 22 23 24
17 18 19 20
13 14 15 16
```

$\phi A \leftrightarrow$

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

16 15 14 13
20 19 18 17
24 23 22 21
```

6.18 The Mask and Mesh Operators

The mask and mesh operators combine the power of the compression and expansion operators in an extremely useful way in that they permit you to construct a new array from the logically selected portions of the other arrays.

6.18.1 The Mask Operator

Let U be a logical vector and suppose X and Y are arrays where $\rho X \leftrightarrow \rho Y$. Then, the U mask of coordinate K of X and Y is written $U/[K](X;Y)$. The resultant array is composed of the elements of coordinate K of Y, corresponding to the ones of U, and the elements of coordinate K of X, corresponding to the zeros of U.

Formally,

$$\rho U/[K](X;Y) \leftrightarrow \rho X \leftrightarrow \rho Y$$
$$U/[K]\ U/[K](X;Y) \leftrightarrow U/[K]Y$$
$$(\sim U)/[K]\ U/[K](X;Y) \leftrightarrow (\sim U)/[K]X$$

Example: Suppose *IORG* ↔ 1 and

$$A \leftrightarrow \begin{matrix} 1 & 3 & 5 & 7 \\ 9 & 11 & 13 & 15 \\ 17 & 19 & 21 & 23 \end{matrix} \qquad B \leftrightarrow \begin{matrix} 2 & 4 & 6 & 8 \\ 10 & 12 & 14 & 16 \\ 18 & 20 & 22 & 24 \end{matrix}$$

Then

$$(1,0,0,1)/[2](A;B) \leftrightarrow \begin{matrix} 2 & 3 & 5 & 8 \\ 10 & 11 & 13 & 16 \\ 18 & 19 & 21 & 24 \end{matrix}$$

6.18.2 The Mesh Operator

Let U be a logical vector and suppose X and Y are arrays where $\rho\rho X \leftrightarrow \rho\rho Y$. Suppose that for some component I, we have $(((\rho\rho X)\rho I)\neq\iota\rho\rho X)/\rho X \leftrightarrow ((\rho\rho Y)\neq\iota\rho\rho Y)/\rho Y$. That is, except for coordinate I, the dimension vectors for X and Y are equal. In addition,

$$(\rho X)[I] \leftrightarrow +/\sim U$$
$$(\rho Y)\ [I] \leftrightarrow +/U$$

Then we define $U\backslash[I](X;Y)$, the U mesh of coordinate I of X and Y as the array of dimension

$$\rho U\backslash[I](X;Y) \leftrightarrow ((I-\underline{IORG})\uparrow(\rho X)),(\rho U),(I+1-\underline{IORG})\downarrow(\rho X)$$

That is, the resultant array consists of the elements of the U expansion of coordinate I of Y and the $\sim U$ expansion of coordinate I of X.

Example: Suppose $\underline{IORG} \leftrightarrow 1$ and A and B are

$$A \leftrightarrow \begin{matrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{matrix} \qquad B \leftrightarrow \begin{matrix} 100 & 200 & 300 & 400 \\ 500 & 600 & 700 & 800 \end{matrix}$$

Then

$$(0,0,1,0,1)\backslash[1](A;B) \leftrightarrow \begin{matrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 100 & 200 & 300 & 400 \\ 9 & 10 & 11 & 12 \\ 500 & 600 & 700 & 800 \end{matrix}$$

6.19 The Rotate Operator

Suppose A is an array and N is a scalar integer. You can rotate

the elements of coordinate I of A cyclically N positions to the left if $N \geq 0$, or N positions to the right if N is negative. This is done by using the dyadic operator ϕ, writing $N\phi A$. A cyclic, left rotation means that the left-most elements migrate around to the right-most position in their coordinate. Since the coordinate is of length $(\rho A)[I]$ the rotation moves elements $(\rho A)[I]|N$ positions to the left or right.

Formally, $N\phi[I]A \leftrightarrow A[[I]((\rho\rho A)\rho \underline{IORG})+(\rho A)[I]|((\rho\rho A)\rho N)+\iota(\rho A+[I]$

Example: If $\underline{IORG} \leftrightarrow 1$ and

$$A \leftrightarrow \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix} \quad \begin{matrix} 10 & 11 & 12 \\ 13 & 14 & 15 \\ 16 & 17 & 18 \end{matrix} \quad \begin{matrix} 19 & 20 & 21 \\ 22 & 23 & 24 \\ 25 & 26 & 27 \end{matrix}$$

Then

$$7\phi[2]A \leftrightarrow 1\phi[2]A \leftrightarrow \begin{matrix} 4 & 5 & 6 \\ 7 & 8 & 9 \\ 1 & 2 & 3 \end{matrix} \quad \begin{matrix} 13 & 14 & 15 \\ 16 & 17 & 18 \\ 10 & 11 & 12 \end{matrix} \quad \begin{matrix} 22 & 23 & 24 \\ 25 & 26 & 27 \\ 19 & 20 & 21 \end{matrix}$$

and

```
                          19 20 21
                          22 23 24
                          25 26 27

                           1  2  3
2φ[1]A←→¯1φ[1]A ←→        4  5  6
                           7  8  9

                          10 11 12
                          13 14 15
                          16 17 18
```

If A is an array such that $\rho\rho A \leftrightarrow {}^{-}1+\rho\rho B$ and $(((\rho\rho B)\rho I)\neq\iota\rho\rho B)/\rho B \leftrightarrow \rho A$ (i.e., the dimension vectors of A and B are in agreement after $(\rho B)[I]$ is suppressed, $A\phi[I]B$ is defined as the rotation of coordinate I of B by A.

Formally, if L is a row of $\underline{\iota}\rho B$, then

$$(A\phi[I]B)[;/L] \leftrightarrow (A[;/(((\rho\rho B)\rho I)\neq\iota\rho\rho B)/L]\phi F\underline{\Delta}B)[C]$$

where $C \leftrightarrow (((\rho\rho B)\rho I)=\iota\rho\rho B)/L$

$$F \leftrightarrow \Phi((((\rho\rho B)\rho I)=\iota\rho\rho B)/(0;1)),(((\rho\rho B)\rho I)=\iota\rho\rho B)/(0;L)$$

Example: Suppose *IORC* ↔ 1 and

```
          2  1  2
A ↔       2  2  3
          1  1  2

          1  2  3
          4  5  6
          7  8  9

         10 11 12
B ↔      13 14 15
         16 17 18

         19 20 21
         22 23 24
         25 26 27
```

Then

```
                7  5  9
                1  8  3
                4  2  6

                16 17 12
A⌽[2]B ↔        10 11 15
                13 14 18

                22 23 27
                25 26 21
                19 20 24
```

and

```
                19 11 21
                22 23  6
                16 17 27

                 1 20  3
A⌽[1]B ↔         4  5 15
                25 26  9

                10  2 12
                13 14 24
                 7  8 18

                 3  1  2
                 5  6  4
                 9  7  8

                12 10 11
A⌽[1]B ↔        15 13 14
                16 17 18

                20 21 19
                23 24 22
                27 25 26
```

6.20 The Catenation of Arrays

In Section 6.9, we defined the catenation of vectors. Two arrays A and B may be catenated along coordinate I provided either:

(1) $\rho\rho A \leftrightarrow \rho\rho B$

or **(2)** $1 \leftrightarrow |(\rho\rho A)-\rho\rho B$

or (3) B or A is a scalar

and (4) the coordinates along which A and B are to be joined are of the same dimension.

The meaning of condition (4) is the following.

(a) If $(\rho\rho A) \leftrightarrow (\rho\rho B)$, then
$((\rho\rho A)\rho I)\neq\iota\rho\rho A)/\rho A \leftrightarrow (((\rho\rho B)\rho I)\neq\iota\rho\rho B)/\rho B$
That is, ρA and ρB are identical for every coordinate, except possibly coordinate I.

(b) If $1 \leftrightarrow (\rho\rho A)-\rho\rho B$, then B is considered as if its dimension vector is $(((\rho\rho A)\rho I)=\iota\rho\rho A)\backslash(1;\rho B)$. This dimension vector must satisfy condition (a).

(c) If $1 \leftrightarrow (\rho\rho B)-\rho\rho A$, then A is considered as if its dimension vector is $(((\rho\rho B)\rho I)=\iota\rho\rho B)\backslash(1;\rho A)$. This dimension vector must satisfy condition (a).

(d) If A or B is a scalar, it is treated as though its dimension vector is identical to that of the nonscalar in all components, except for a unit component.

Then the catenation along coordinate I of B and A, written $A,[I]B$, is the array C such that

$$C[[I]\iota(\rho A)[I]]\leftrightarrow A$$
$$C[[I](\rho A)[I]+(\rho B)[I]]\leftrightarrow B$$

Example: If *IORG* $\leftrightarrow$ 1 and

```
      1  3  5  7
A ↔   9 11 13 15

      2  4  6  8
B ↔  10 12 14 16
```

Then

```
                 1  3  5  7
A,[1]B ←→        9 11 13 15
                 2  4  6  8
                10 12 14 16

                 1  3  5  7  2  4  6  8
A,[2]B ←→        9 11 13 15 10 12 14 16
```

If

```
                50 51 52 53
                54 55 56 57
C ←→
                58 59 60 61
                62 63 64 65
```

Then

```
                 1  3  5  7
                 9 11 13 15

A,[1]C ←→       50 51 52 53
                54 55 56 57

                58 59 60 61
                62 63 64 65
```

```
                 1  3  5  7
A,[1]2 ←→        9 11 13 15
                 2  2  2  2
```

```
                 1  3  5  7 2
A,[2]2 ←→        9 11 13 15 2
```

6.21 The Lamination of Arrays

When two arrays, *A* and *B*, are laminated together on coordinate *I*, a new coordinate that has indices ⍳2 is formed before coordinate *I*. The argument *A* fills the first index of the new coordinate; the argument *B* fills the second index of the new

coordinate. The notation is $A:[I]B$, where either $\rho A \leftrightarrow \rho B$ or A or B is a scalar and I is an element of $\iota 1+((\rho\rho A)\lceil\rho\rho B)$ If A or B is a scalar, it is considered as if it were reshaped to have the same shape as the nonscalar argument.

Example: Suppose $\underline{IORG} \leftrightarrow 1$ and A and B are the same as in Section 6.20.

Then

$A:[1]B \leftrightarrow$

```
 1  3  5  7
 9 11 13 15
 2  4  6  8
10 12 14 16
```

$A:[2]B \leftrightarrow$

```
 1  3  5  7
 2  4  6  8
 9 11 13 15
10 12 14 16
```

$A:[3]B \leftrightarrow$

```
 1  2
 3  4
 5  6
 7  8
 9 10
11 12
13 14
15 16
```

$A:[1]5 \leftrightarrow$

```
 1  3  5  7
 9 11 13 15
 5  5  5  5
 5  5  5  5
```

6.22 The Dyadic Transposition of Arrays

The dyadic transposition of an array is a generalization of a monadic transposition of an array. The V transpose of A is written $V\lozenge A$, where V is a vector containing one element for each coordinate of A.

The values of the elements of V indicate the dimension of the resulting array. If you want the result R to be of rank $\rho\rho R$, V must contain, in any order, at least one of each of the elements from $\iota\rho\rho R$. Further, the elements of V are limited to the values contained in $\iota\rho\rho R$.

If the same value appears more than once in V, which must occur if $(\rho\rho R)<\rho V$, then the repeated dimension of R is to be formed from more than one dimension of A. For example, suppose that $\rho\rho A \leftrightarrow 4$, $\underline{IQRG} \leftrightarrow 1$, and $V \leftrightarrow (2,1,2,2)$. The result $R \leftrightarrow V\lozenge A$ must be an rank-2 array since $2 \leftrightarrow \lceil/V$. The first dimension of R is formed from the second dimension of A. The second dimension of R is formed from the first, third and fourth dimensions of A. Thus, the elements to be selected from A are of the form $A[I;J;I;I]$, where I and J are scalar integers. This is the diagonal passing through $A[1;1;1;1]$ and through the first, second and fourth dimensions of A. The diagonal contains no more elements than the shortest of the dimensions from which it is taken.

In summary, the vector V of positive integers must satisfy the conditions:

(1) $\rho V \leftrightarrow \rho\rho A$. There must be an element in V for every component of A.

(2) Every element of V must be an element of $\iota\rho\rho A$.

(3) Every element of $\iota\lceil/V$ must be an element of V. (For example, if $\rho\rho A \leftrightarrow 6$ and the largest element in V is 5, V must contain all of the elements of $\iota 5$. Since V must have six components, one of the elements of $\iota 5$ must be repeated.)

Then, the transpose of A by V is defined as:

(a) $\rho\rho V\lozenge A \leftrightarrow 1+(\lceil/V)-\underline{IORG}$

(b) For each element I of $\iota\rho\rho V\lozenge A$,
$\rho(V\lozenge A)[I] \leftrightarrow \lfloor/(V=(\rho V)\rho I)/\rho A$

(c) For each row L of $\iota\rho V\lozenge A$,
$(V\lozenge A)[;/L \leftrightarrow A[;/L[V]]A$

In order to better understand dyadic transposition, study the following detailed examples.

Examples: Suppose $\underline{IORG} \leftrightarrow 1$,
$\rho A \leftrightarrow (5,7,3,8)$,
and $V \leftrightarrow (2,1,2,2)$.
Then if $R \leftrightarrow V\lozenge A$,
$\rho\rho R \leftrightarrow 2$.
From property (b), we see that

$(\rho R)[1] \leftrightarrow \lfloor/((2,1,2,2)=(1,1,1,1))/(5,7,3,8) \leftrightarrow \lfloor/(0,1,0,0)/(5,7,3,8) \leftrightarrow 7$
$(\rho R)[2] \leftrightarrow \lfloor/((2,1,2,2)=(1,1,1,1))/(5,7,3,8) \leftrightarrow \lfloor/(5,3,8) \leftrightarrow 3$

Therefore, $(\rho R) \leftrightarrow (7,3)$

From property (c), we can determine the mapping between elements of A and R.

```
(V⍀A)[1;1] ↔ A[;/(1,1)[2 1 2 2]] ↔ A[1;1;1;1]
(V⍀A)[1;2] ↔ A[;/(1,2)[2 1 2 2]] ↔ A[2;1;2;2]
(V⍀A)[1;3] ↔ A[;/(1,3)[2 1 2 2]] ↔ A[3;1;3;3]
(V⍀A)[2;1] ↔ A[;/(2,1)[2 1 2 2]] ↔ A[1;2;1;1]
                    . . .
(V⍀A)[5;2] ↔ A[;/(5,2)[2 1 2 2]] ↔ A[2;5;2;2]
                    . . .
(V⍀A)[7;3] ↔ A[;/(7,3)[2 1 2 2]] ↔ A[3;7;3;3]
```

Thus, you see that the elements of $V⍀A \leftrightarrow A[I;J;I;I]$, **where** J **is an element of** $\iota 7$ **and** I **is an element of** $\iota 3$.

Suppose we want $X \leftrightarrow W⍀A$ where $W \leftrightarrow (3,1,2,2)$ Then $\rho\rho \leftrightarrow 3$ **and** $\rho X \leftrightarrow (7,3,5)$. **The mapping between** A **and** X **is given by**

$$X[I;J;K] \leftrightarrow A[K;I;J;J]$$

where I **is an element of** $\iota 7$, J **is an element of** $\iota 3$, **and** K **is an element of** $\iota 5$.

Finally, suppose

```
      1  2  3
      4  5  6
      7  8  9
     10 11 12
B ↔
     13 14 15
     16 17 18
     19 20 21
     22 23 24
```

Verify that the following are true:

(see next page)

```
(1,1,1)⍉B ↔ (1,13)

                1  2  3
(1,1,2)⍉B ↔ 13 14 15

                1  4  7 10
(1,2,1)⍉B ↔ 13 16 19 22

               1 13
(2,1,1)⍉B ↔    5 17
               9 21

                1  5  9
(1,2,2)⍉B ↔ 13 17 21

               1 14
(2,1,2)⍉B ↔    4 17
               7 20
              10 22

               1 16
(2,2,1)⍉B ↔    2 17
               3 18

(1,2,3)⍉B ↔ B

                1  4  7 10
                2  5  8 11
                3  6  9 12
(1,3,2)⍉B ↔
               13 16 19 22
               14 17 20 23
               15 18 21 24
```

```
                    1 13
                    2 14
                    3 15

                    4 16
                    5 17
                    6 18
(3,1,2)⍉B ↔
                    7 19
                    8 20
                    9 21

                   10 22
                   11 23
                   12 24
```

```
                    1 13
                    4 16
                    7 19
                   10 22

                    2 14
                    5 17
(3,2,1)⍉B ↔         8 20
                   11 23

                    3 15
                    6 18
                    9 21
                   12 24
```

```
                    1  2  3
                   13 14 15

                    4  5  6
                   16 17 18
(2,1,3)⍉B ↔
                    7  8  9
                   19 20 21

                   10 11 12
                   22 23 24
```

```
                    1  4  7 10
                   13 16 19 22

(2,1,1)⍉B ↔        2  5  8 11
                   14 17 20 23

                    3  6  9 12
                   15 18 21 24
```

CHAPTER SEVEN

EXPRESSIONS, STATEMENTS AND PROGRAMS

So far, you have seen a number of APPLE's arithmetic and manipulative operators. You have nearly enough information to write a program. However, you need to learn about APPLE statements and expressions before you can write a program.

In this chapter, we will cover everything you need to write a simple program using APPLE operators. In the following chapters, you will be introduced to operations on arrays, as well as how to define your own functions and operators.

7.1 Elementary Definitions

An expression is any well-defined combination of operators and operands. Expressions always have a value.

A statement is an expression whose value is the empty vector. Statements always perform some action. In FORTRAN, they include, for example, assignment statements, DO statements, IF statements, and so forth.

7.2 Conformability Conventions for Scalar Dyadic Operators

Let us reëxamine the simple expressions $A*B$ where $*$ is some scalar dyadic operator. Until now, we have stated that this expression is defined only when $\rho A \leftrightarrow \rho B$. Two such arrays are said to be conformable. The result is an array C, where $\rho C \leftrightarrow \rho A \leftrightarrow \rho B$ and for any subscript L from $\iota\rho C$, we have

$$C[;/L] \leftrightarrow A[;/L]*B[;/L]$$

We will now extend the definition of conformability with respect to a scalar dyadic operator.

First, suppose only one of the two operands, operand A, is a one-element array. (A is a one-element array if and only if $1 \leftrightarrow \times/\rho A$. Thus, A might be a scalar or a one-element vector.) We subsequently define $A*B$ to be the array $C \leftrightarrow ((\rho B)\rho A)*B$. (If B was the one-element array, we would have $C \leftrightarrow A*(\rho A)\rho B$) That is, the scalar operand is applied to every element of the non-scalar operand.

Example: If $IORG \leftrightarrow 1$, then

$1+\iota 5 \leftrightarrow (2,3,4,5,6)$

If

$$A \leftrightarrow \begin{matrix} -1 & 3 & 4 \\ 2 & -7 & 8 \end{matrix}$$

Then

$$A\times -2 \leftrightarrow \begin{matrix} 2 & -6 & -8 \\ -4 & 14 & 16 \end{matrix}$$

If both operands are one-element arrays, then $A*B$ is defined to be the array C where $C \leftrightarrow ((\rho\rho A)\lceil\rho\rho B)\rho(,A)*,B$. That is, the rank of the resultant one-element array equals the maximum of the ranks of the operands.

If neither of the operands is a one-element array, the arrays are conformable only when the two arrays satisfy one of the following two conditions.

(1) Assume that $(\rho\rho A)>\rho\rho B$ and $(-\rho\rho B)\uparrow\rho A \leftrightarrow \rho B$. That is, the last $\rho\rho B$ elements of ρA are identical to ρB. (If $(\rho\rho B)>\rho\rho A$, everything works when you mentally interchange their names.) Then, the arrays are conformable and $C \leftrightarrow A*(\rho A)\rho B$

(2) If (1) is not true, but there are non-negative scalar integers M and N such that $M<\rho\rho A$ and $N<\rho\rho B$ and

$$M\downarrow\rho A \leftrightarrow N\downarrow\rho B$$

Then A and B are conformable only if $M\uparrow\rho A \leftrightarrow M\rho 1$ or $N\uparrow\rho A \leftrightarrow M\rho 1$. The result is of rank $\rho\rho C \leftrightarrow (\rho\rho A)\lceil\rho\rho B$, $\rho C \leftrightarrow (\phi(\rho\rho C)\rho(\phi\rho A),D\rho 1)\lceil\phi(\rho\rho C)\rho(\phi\rho B),D\rho 1$ and $C \leftrightarrow ((\rho C)\rho A)*(\rho C)\rho B$, where $D \leftrightarrow |(\rho\rho A)-\rho\rho B$.

In all other cases, A and B are non-conformable arrays.

Examples: If $\rho A \leftrightarrow (2,3,5)$ and $(\rho B) \leftrightarrow (1,1,1,2,3,5)$ then A and B are conformable and $\rho A*B \leftrightarrow (1,1,1,2,3,5)$. If $\rho E \leftrightarrow (1,4,3,2,8)$ and $\rho F \leftrightarrow (1,1,6,4,3,2,8)$, then E and F are conformable and $\rho E*F \leftrightarrow (1,1,6,4,3,2,8)$. If $\rho G \leftrightarrow (1,2,3,5,6,2,8)$ and $\rho H \leftrightarrow (1,1,1,1,1,1,1,1,2,8)$.

Then G and H are conformable and
$\rho G \ast H \leftrightarrow (1,1,1,1,2,3,5,6,2,8)$.

If $\rho V \leftrightarrow (1,2,3,4)$ and $\rho W \leftrightarrow (1,3,3,4)$, V and W are <u>not</u> conformable. (Why?)

Up until now, we were very careful to write expressions like $(I=\sim\rho\rho A)$ as $((\rho\rho A)\rho I)=\sim\rho\rho A$, so that the conformity requirements were trivially satisfied. For the remainder of this tutorial, we can use the conformity conventions just described.

7.3 Specification Expressions

If A is a variable and you want to store the value of an expression E into A, you use the specification operator $\leftarrow$ and write

$A \leftarrow E$

By definition, the shape of A will equal the shape of E, i.e., $\rho A \leftrightarrow \rho E$. Since $\leftarrow$ is a dyadic operator, you may use it anywhere you would use any other dyadic operator. The value of $A \leftarrow E$ is the new value of A.

7.3.1 Select Expressions and Specifications

Let E be any well-formed, array-valued expression. Then, F is a <u>select expression</u> on E if it is a well-formed expression

consisting of an arbitrary number of the following operators applied to E.

(1) take

(2) drop

(3) reversal

(4) rotate

(5) subscripting

(6) Δ

(7) $\underline{\Delta}$

(8) compression

(9) expansion

(10) mesh

(11) mask

(12) transposition

Then, if F is a select expression on the previously specified array *A*, when you write (F)←(E). If (ρF)>ρE the specification is equivalent to *A*←F←*E*. Another way of looking at this concept is: the left-hand side of a specification may be any selection expression on *A* that could have been written as a subscript expression on *A*.

Examples: Suppose *IORG* ↔ 1 and *A* has been specified as

A ← (2,3)ρι6. Then

```
A[1 2 ;3] ← ι2
```

produces

```
      1  2  1
A ↔   4  5  2
```

This could also have been written as either

```
      ((0,0,1)/A) ← ι2
or          (¯1↑A) ← ι2
```

We also can write

```
      (⍉A) ← (3,2)ρ6+ι6
```

producing

```
          7  9 11
A ←→ 8 10 12
```

7.4 Conditional Statements

APPLE, like many programming languages, contains conditional (or "IF") statements. A single conditional statement always contains some test (i.e., an expression that evaluates to a logical scalar). If the test is satisfied (evaluates to 1), then the expression associated with the conditional statement is evaluated. If the test fails (evaluates to 0), then control is transformed to the statement immediately following the conditional statement.

A more intricate conditional statement consists of a test with its associated expression and an ordered sequence of alternatives. If the test succeeds, its associated expression is evaluated and control is transferred to the statement immediately following the entire conditional statement. If the test fails, each of the alternative tests is executed until either some one of them is satisfied or they all fail. As soon as the first alternative test is satisfied, its associated

expression is evaluated and control is transferred to the statement immediately following the entire conditional statement.

7.4.1 One-Line Conditional Statements

The simplest kind of conditional statement fits on one line. It is of the form

IF test *THEN* expression

There is also a version with an alternative. This takes the form

IF test *THEN* expression 1 *ELSE* expression 2

In the conditional statement, if the test is true, then expression 1 is evaluated and control transfers to the statement immediately following. If the test is false, expression 2 is evaluated and control is transferred to the statement immediately following.

Examples:

```
IF A>10 THEN X ← +/Y
```

If $1 \leftrightarrow A>10$ then X is redefined as $+/Y$.
Otherwise X retains its original value.

```
IF A >10 THEN X ← +/Y ELSE X ← ⌈/Y
```

This time X will be changed regardless of the value of $A>10$.

The statement is equivalent to `X ← (A>10)/(⌈/Y;+/Y)`

7.4.2 Multi-line Conditional Statements

If a conditional statement will not fit on one line, it is necessary to use a multi-line conditional statement. This form is far more powerful than the one-line conditional statement, which cannot control the conditional execution of a set of statements.

The simplest type of multi-line conditional statement involves only one test and has no alternatives. It is of the form

```
IF test
        statement 1
        statement 2
             .
             .
             .
        statement n
ENDIF
```

Here, if the test is true, statement 1,..., statement n are all executed in order. If the test is false, control is passed to the statement immediately following the associated *ENDIF*.

There is also a multi-line conditional statement with an alternative.

It is of the form

```
IF  test
        s1
        s2
         .
         .
         .
        sn
ELSE    t1
        t2
         .
         .
         .
        tm
ENDIF
```

If the test is true, statements s1,...,sn are executed and control transfers to the statement immediately following the associated *ENDIF*. If the test is false, then statements t1,...,tm are executed and control is transferred to the statement immediately following the *ENDIF*.

The most general type of multi-line conditional statement allows you to write as many conditional alternatives as you need and an *ELSE* alternative if you want one.

It is of the form

```
IF   test 1
          s1
          s2
            .
            .
            .
          sn
OR IF test 2
          t1
          t2
            .
            .
            .
          tm
OR IF test 3
          u1
            .
            .
            .
          ul
  .
  .
  .
ELSE      v1
            .
            .
            .
          vk
ENDIF
```

First, if test 1 is true, then statements s1,...,sn are evaluated, and control is passed to the statement immediately following the associated *ENDIF*. If test 1 is false, test 2 is evaluated. If test 2 is true, statements t1,...,tm are evaluated and control is passed to the statement immediately following the associated *ENDIF*. This process continues until either a test is true and its associated statements are executed, or until all of the alternatives have been exhausted. If the last alternative is an *ELSE*, its associated statements

will be executed if all the preceding tests have failed.

You should note that any of the statements associated with a test or an *ELSE* can, itself, be a conditional statement. If such a conditional statement is executed, it is treated exactly as if it was a conditional statement occuring elsewhere in a program: it will either be a one-line conditional statement, or there will be an *ENDIF* associated with it.

Example:

```
IF (A<B)∨ C>D ← Q+R
     X←+/Y
     Z←⍉A
OR IF (A>B)∧D≠0
     X ← A⍠D
     IF Q>R
          Z ← V⍉R
     ELSE Z ← (⍎V)⍉R
     ENDIF
ELSE Z ← (⍴Z)⍴0
ENDIF
```

In this example, *D* is specified in the first test. Regardless of the truth of that test, *D* will retain the value *Q+R* until it is respecified elsewhere. The *OR IF* alternative, which will be executed only if the first test fails, contains a conditional statement of its own. That statement determines the value of *Z*. If both tests fail, *Z* is respecified as an array of zeros.

You should indent conditional statements the way we have done in this example. When you do, it is very easy to identify the statements associated with each alternative. This is especially true when a conditional statement contains other conditional statements nested within it.

7.5 Conditional Expressions

The right-hand side of any specification may be a conditional expression. A conditional expression looks exactly like a conditional statement, except that there is now a value associated with it. This is because every APPLE statement is an expression the value of which is discarded.

When a conditional statement becomes a conditional expression, its value is the value of the last expression in the alternative that is executed. If it is possible that none of the alternatives be executed, you must be sure to include an *ELSE* alternative, otherwise the value of the conditional expression will be undefined.

The shape of the value of a conditional expression is determined by the shape of the last expression in each of its alternatives. This shape is determined exactly the way it would be determined if these expressions were to be operated on by some dyadic scalar operator. That is, these expressions must be pair-wise

conformable. Therefore, the shape of the result of a conditional expression is the conformed shape of the last expression of each alternative of the conditional expression.

Example: You can calculate the value of !*V*,

where *N* is a nonnegative scalar integer by writing

```
FACTORIAL ← IF N=0 THEN 1 ELSE ×/⍳N+1-IORG
```

7.6 Iteration Statements

Iteration statements are akin to the DO-loops of FORTRAN. They provide a convenient means of performing the same set of calculations repeatedly on some set of elements.

7.6.1 The DO Statement

This statement is of the form

```
DO I ∈ V
    s1
    s2
    .
    .
    .
    sn
REPEAT
```

where *I* is a variable name, *V* is an array-valued expression, and s1,s2,...,sn are statements. Statements s1,...,sn will be executed together (⍴*V*)[*IORG*] times, each time with *I* assuming

one of the values

$$V[[\underline{IORG}]\underline{IORG}],V[[\underline{IORG}]1+\underline{IORG}],\ldots,V[[\underline{IORG}]\lceil/\iota(\rho V)[\underline{IORG}]]$$

Thus, if $\underline{IORG} \leftrightarrow 1$ and you write

```
DO U≤ 6×ι5
     A[I] ← 2×I
REPEAT
```

Then, this is equivalent to your having written

```
 A[6] ← 12
A[12] ← 24
A[18] ← 36
A[24] ← 48
A[30] ← 60
```

Of course, you could have written $A[6\times\iota5] \leftarrow 2\times I$ in this case, but that is because the body of the loop only contained one statement. Since the body of the loop can contain arbitrarily many statements of arbitrary complexity, you could write, e.g.,

```
DO J≤ ⍳⍴B
     F ← ⍉((I=ι⍴⍴B)/(0;1)), (I=ι⍴⍴B)/(0;J)
     (F⍙Z) ← A[;/(I≠ι⍴⍴B)/J]⌽F⍙B
REPEAT
```

In this code, B is an arbitrary array, and J successively takes on the value of each row of $\iota\rho B$. If the dimension of B is unknown at coding time, this loop could not be written as a sequence of statements without use of some form of iteration.

7.6.2 WHILE Statements

The *WHILE* statement is of the form

```
WHILE test
      s1
      s2
      .
      .
      .
      sn
REPEAT
```

The test is any expression that evaluates to a logical scalar, and the s1,...,sn are statements.

The test is evaluated and, if true, statements s1,...,sn are evaluated. Then, the test is reevaluated. If it is true, statements s1,...,sn are evaluated again. This process continues until the test is false, at which time control is transferred to the statement immediately following the *REPEAT*.

The *WHILE* statement is useful for controlling some process that must iterate while some condition is satisfied, e.g., a numerical approximation process. Note that iteration continues as long as the test is true. Consequently, you must provide a means for either causing the test to eventually evaluate to false, or make use of one of the operators described in Section 7.6.4.

Example:

```
WHILE ∧/EPSILON≤|X-Y
      X ← (|X-1⌽X)÷2
      Y ← (|Y+(|⌽Y)+W(¯1⌽Y))÷3
REPEAT
```

will be iterated until every component of the absolute value of *X-Y* is less than *EPSILON*.

7.6.3 UNTIL Statements

UNTIL statements are written

```
UNTIL test
      s1
      s2
      .
      .
      .
      sn
REPEAT
```

where the test is any expression evaluating to a logical scalar, and s1,...,sn are statements.

The sequence of statements is iterated until the test is true. For a given test *T*, the loop *UNTIL T* is equivalent to the loop *WHILE~T*. Two kinds of conditional loops are provided to permit programmers who think in terms of a termination condition, rather than in terms of a continuation condition, to directly translate their thought process into APPLE code.

7.6.4 Iteration Control Operators

It is often necessary to prematurely terminate one or more iterations of a loop because some boolean condition is satisfied. APPLE provides you with three anadic operators, [i.e., having no operands] with which you can direct the flow of control within such loops.

> The first of these operators is *CYCLE*, which terminates execution of the current iteration and transfers control to the top of the loop for the next iteration.
>
> The *LEAVE* operator causes control to be transferred to the statement immediately following the *REPEAT* associated with the loop. This causes the loop to definitively stop iterating.
>
> The *EXIT* operator causes control to be transferred to the statement immediately following the *REPEAT* associated with the outermost loop in which the instruction occurs.

If there are no nested loops, *EXIT* and *LEAVE* are equivalent and either operator may be used. But, if *EXIT* is encountered in a nested body of loops, then all of the loops stop iterating immediately.

7.7 The CASE Statement

The *CASE* statement is a simplification of the conditional statement. It is used when you want to evaaluate one of a set of expression sequences based on the value of some scalar expression, usually integer-valued.

The *CASE* statement takes the form

```
    CASE scalar expression
    value list 1 → statement sequence 1
    value list 2 → statement sequence 2
                    .
                    .
                    .
    value list n → statement sequence n
   ELSE statement sequence
   ENDC
```

The *ELSE* clause is optional. If together, the value lists exhaust the possible values of the scalar expression, you do not have to provide an *ELSE* clause.

The value list is a sequence of scalar constants, separated from each other with semicolons. The statement sequences may contain any combination of APPLE statements, including conditional or iteration statements.

Example. This *CASE* statement computes $!N$ and stores it in Z.

N and Z are assumed to be one-element, nonnegative scalar integers.

```
CASE IORG

0 → Z ← ×/1+ιN
1 → Z ← ×/ιN
ENDC
```

We could also write this code sequence using two *CASE* statements.

```
CASE N

0;1 → Z ← 1
ELSE  CASE IORG
        0 → Z ← X/1+ιN
        1 → Z ← X/ιN
      ENDC
ENDC
```

Of course, this could have been more concisely written as `Z ← ×/IORG+ιN`, but our intent was only to illustrate the use of the *CASE* statement in an elementary context.

7.8 The CASE Expression

The *CASE* statement is a simplification of the conditional statement, so any *CASE* statement can be rewritten as a conditional statement. There is a *CASE* expression in the APPLE language. The same conformability conventions apply to *CASE* expressions as apply to conditional expressions.

Example: We could rewrite the second example of Section 7.7 as:

```
Z ← CASE N
    0;1 → 1
    ELSE CASE IORG
           0 →  ×/1+⍳N
           1 →  ×/⍳N
         ENDC
    ENDC
```

CHAPTER EIGHT

ARRAY OPERATIONS

So far, we have investigated scalar operations on conformable arrays, and manipulative operations that produce subarray and permutations of the elements of arrays. Your background is now sufficiently strong in APPLE so that we can consider the class of operators that perform numerical manipulations on arrays.

8.1 The Index of an Array Within an Array

Suppose A is some vector or a one-component array and B is an arbitrary array. Then $A\iota B$, the index in A of B is an array such that

$$\rho A\iota B \leftrightarrow \rho B$$

and for each row L of $\iota\rho B$, $(A\iota B)[;/L]$ is the least index I such that $(,A)[I] \leftrightarrow B[;/L]$. If $B[;/L]$ is not an element of A, then $(A\iota B)[;/L] \leftrightarrow 1+\lceil/\iota\rho,A$.

Example: If $\underline{LORG} \leftrightarrow 1$, $A \leftrightarrow (3,\bar{}7,2,0,1,4,1)$ and

$$B \leftrightarrow \begin{matrix} 1 & 3 & 2 \\ \bar{}7 & 4 & 5 \end{matrix}$$

$$B \leftrightarrow \begin{matrix} 0 & \bar{}1 & 4 \\ 3 & \bar{}2 & 1 \end{matrix}$$

Then

$$A\iota B \leftrightarrow \begin{matrix} 5 & 1 & 3 \\ 2 & 6 & 8 \end{matrix}$$

$$A\iota B \leftrightarrow \begin{matrix} 4 & 5 & 6 \\ 1 & 8 & 5 \end{matrix}$$

Note that $A[5] \leftrightarrow A[7] \leftrightarrow 1$, but the index returned to $A\iota B$ is always 5 since 5 is the least index I for which $A[I] \leftrightarrow 1$. Note also that an index of 8 was returned for the two elements 5 and $\bar{}2$, which are not contained in A.

You can tell if every element of B is an element of A since, if that is the case, we must have

$$1 \leftrightarrow \vee/(,A\iota B) \leq \lceil/\iota\rho A$$

8.2 Array Membership

Let A and B be two arbitrary arrays. Then, $A \in B$ is a logical array of the same rank as A and contains a 1 corresponding to each element of A that is present somewhere in B.

Example: Suppose A and B are the arrays from the example in Section 8.1.

Then

$$A \in B \leftrightarrow (1,1,1,1,1,1,1)$$

and

$$B \in A \leftrightarrow \begin{matrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{matrix}$$

8.3 Sorting

If A is a vector, then $\grade{up}A$ is a vector, $\rho \grade{up}A \leftrightarrow \rho A$, such that $(\grade{up}A)[I]$ is the index of the I-th smallest element of A. That is, $A[\grade{up}A]$ is a vector whose first element is the least element of A and each of whose remaining elements is no less than its predecessor element.

Just as $\grade{up}A$ can be used to sort the elements of A into ascending order, $\grade{down}A \leftrightarrow \phi \grade{up}A$ can be used to sort A into descending order.

Examples: If *IORG* ↔ 1 and *A* ↔(6,3,5,3,9,3,1)

Then

⍋*A* ↔ (7,2,4,6,3,1,5)

⍒*A* ↔ (5,1,3,6,4,2,7)

8.4 Outer Products

Suppose *A* and *B* are any two arrays and ⍟ is any scalar dyadic operator. The ⍟ outer product of *A* and *B*, written *A*∘.⍟*B*, is an array containing the ⍟ product between every element of *A* and every element of *B*.

Formally, ⍴*A*∘.⍟*B* ↔ (⍴*A*),⍴*B* and for each row *L* of ⍳⍴*A*∘.⍟*B*,

(*A*∘.⍟*B*)[;/*L*] ↔ *A*[;/(⍴⍴*A*)↑*L*]⍟*B*[;/(-⍴⍴*B*)↑*L*]

Examples: Suppose $A \leftrightarrow (1,2,3,4,5,6,7)$

Then

```
                1  2  3  4  5  6  7
                2  4  6  8 10 12 14
                3  6  9 12 15 18 21
A∘.×A ←→        4  8 12 16 20 24 28
                5 10 15 20 25 30 35
                6 12 18 24 30 36 42
                7 14 21 28 35 42 49

                1 0 0 0 0 0 0
                0 1 0 0 0 0 0
                0 0 1 0 0 0 0
A∘.=A ←→        0 0 0 1 0 0 0
                0 0 0 0 1 0 0
                0 0 0 0 0 1 0
                0 0 0 0 0 0 1

                1 1 1 1 1 1 1
                0 1 1 1 1 1 1
                0 0 1 1 1 1 1
A∘.≤A ←→        0 0 0 1 1 1 1
                0 0 0 0 1 1 1
                0 0 0 0 0 1 1
                0 0 0 0 0 0 1

                                0  6  6
                                0 12 12
                                0 45 13
((2,3)ρ(6,12,45,8,¯5,2))∘.⌊(0,67,13) ←→
                               ¯0 ¯8 ¯8
                               ¯5 ¯5 ¯5
                                0  2  2
```

8.5 Inner Products

The APPLE inner product is a generalization of the linear algebra inner product of two matrices. Suppose that A and B are

two matrices such that $^{-}1\uparrow\rho A \leftrightarrow 1\uparrow\rho B$. The following code produces the inner product C of A and B.

```
C←((1↑ρA),¯1↑ρB)ρ0
DO L ≤ ⊥ρC
     C[;/L]←+/A[1↑L;]×B[;1↓L]
REPEAT
```

(In normal parlance, $C[I;J]$ is the sum of the componentwise product of the I-th row of A with the J-th column of B.)

The APPLE notation for this inner product is $A+.\times B$.

In general, if A and B are matrices satisfying $^{-}1\uparrow\rho A \leftrightarrow 1\uparrow\rho B$ and $*$ and $\circledast$ are two scalar dyadic operators from Table III (see Section 5.3), then the $*$-$\circledast$ inner product of A and B is written $A*.\circledast B$, where $\rho A*.\circledast B \leftrightarrow (1\uparrow\rho A),^{-}1\uparrow\rho B$ and for any row L of $\perp\rho A*.\circledast B$

$$A*.\circledast B[;/L] \leftrightarrow *\/A[1\uparrow L;]\circledast B[;1\downarrow L]$$

Examples:

```
1 2       1 2 3       9 12 15
3 4 +.×   4 5 6  ↔   19 26 33
5 6                   29 40 51
```

```
1 2       1 2 3       5  7  9
3 4 +.⌈   4 5 6  ↔    7  8  9
5 6                   11 11 11
```

In the remainder of this section we generalize the inner product to conformable arrays of arbitrary rank.

8.5.1 Inner Product Conformability

If $\rho\rho A \leftrightarrow \rho\rho B \leftrightarrow 0$ then $A \divideontimes . \times B \leftrightarrow \divideontimes / A \times B$.

Otherwise, $A \divideontimes . \times B$ is defined only if A and B satisfy one of the following conditions:

(1) $\rho\rho A \leftrightarrow 0$
(2) $\rho\rho B \leftrightarrow 0$
(3) $^{-}1\uparrow\rho A \leftrightarrow 1\uparrow\rho B$
(4) $1\uparrow\rho B \leftrightarrow 1$
(5) $^{-}1\uparrow\rho A \leftrightarrow 1$

8.5.2 Definition of the General Inner Product

If the conformability conditions are satisfied, then A and B are conceptually replaced by arrays $\underline{A}$ and $\underline{B}$ given by:

(1') If $\rho\rho A \leftrightarrow 0$ then
$\underline{A} \leftrightarrow (1\uparrow\rho B)\rho A$
$\underline{B} \leftrightarrow B$

(2') If $\rho\rho B \leftrightarrow 0$ then
$\underline{A} \leftrightarrow A$
$\underline{B} \leftrightarrow (^{-}1\uparrow\rho A)\rho B$

(3') If $^{-}1\uparrow\rho A \leftrightarrow 1\uparrow\rho B$ then

$\underline{A} \leftrightarrow A$
$\underline{B} \leftrightarrow B$

(4') If $1\uparrow\rho B \leftrightarrow 1$ then

$\underline{A} \leftrightarrow A$
$\underline{B} \leftrightarrow ((^{-}1\uparrow\rho A), 1\downarrow\rho B)\rho B$

(5') If $^{-}1\uparrow\rho A \leftrightarrow$ then

$$\underline{A} \leftrightarrow (\phi\iota\rho\rho A)\circledast((Q\uparrow\rho B),\phi^{-}1\downarrow\rho A)\rho(\phi\iota\rho\rho A)\circledast A$$
$$\underline{B} \leftrightarrow B$$

Then $A\overset{*}{.}\times B \leftrightarrow \underline{A}\overset{*}{.}\times\underline{B} \leftrightarrow C$ where

$$\rho C \leftrightarrow (^{-}1\downarrow\rho\underline{A}),1\downarrow\rho\underline{B}$$

and for each row L of $\perp\rho C$

$$C[;/L] \leftrightarrow *\!/(G\Delta\underline{A})\times H\Delta\underline{B}$$

where

$$G[;\lfloor/\iota 2] \leftrightarrow ((^{-}1+\rho\rho\underline{A})\rho 1),0$$
$$G[;\lceil/\iota 2] \leftrightarrow ((^{-}1+\rho\rho\underline{A})\uparrow L),0$$
$$H[;\lfloor/\iota 2] \leftrightarrow 0,(^{-}1+\rho\rho\underline{B})\rho 1$$
$$H[;\lceil/\iota 2] \leftrightarrow 0,(1-\rho\rho\underline{B})\uparrow L$$

Example:

```
                          ¯3 ¯5
                          ¯7 ¯9

1 2       1 2          ¯1  ¯1
3 4       3 4          ¯3  ¯5
    +.-       ↔
5 6       5 6           5   3
7 8       7 8           1  ¯1

                          9   7
                          5   3
```

8.6 Change of Base

The number 2307 = $2\times10^3+3\times10^2+0\times10^1+7\times10^0$. This is a polynomial representation of the number 2307 in the radix 10. We also have $2307=3\times5^4+3\times5^3+2\times5^2+1\times5^1+2\times5^0$ as a polynomial representation of

2307 in the radix 5.

If you have a vector representation V of a number N in the base B, you can compute the value of N by evaluating the expression $(\times\backslash((^{-}1+\rho V)\rho B),1)+.\times V$. For example, if $B \leftrightarrow 5$ and $V \leftrightarrow (3,3,2,1,2)$, the expression evaluates to

$$(\times\backslash(5,5,5,5,1))+.\times(3,3,2,1,2) \leftrightarrow (625,125,25,5,1)+.\times(3,3,2,1,2$$
$$\leftrightarrow 1875+375+50+7 \leftrightarrow 2307$$

The APPLE notation for this conversion is $B\perp V$ (read "B decode V").

Conversely, if you want the vector representation V of N to the base B, then you can compute V by the Euclidian Algorithm as follows:

```
V ← ¯ϵ
UNTIL N=0
      V←(B|N),V
N←⌊N÷B
REPEAT
```

The APPLE notation for this inverse conversion is $B\top N$ (read "B encode N").

Change of base can be generalized to cover mixed bases. If B is a vector, then $B\perp V$ is defined when B and V are conformable for inner product:

$$B\perp V \leftrightarrow (\times\backslash(1\downarrow B),1)+.\times V$$

Indeed, $B\perp V$ is similarly defined for any arrays B and V that are conformable under inner product. The formal definition is extremely complicated, so let us look, instead, at a few illustrative examples.

Examples: Suppose you want to know the number of seconds in 1 year, 20 days, 5 hours, 3 minutes and 17 seconds. We will proceed with the assumption that 1 year=365 days, 1 day=24 hours, 1 hour=60 minutes, and 1 minute=60 seconds. Then we would have to compute

```
(365×24×60×60),(24×60×60),(60×60),60,1)+.×(1,20,5,3,17).
```

But this is equivalent to

```
(×\(365,24,60,60,1))+.×(1,20,5,3,17)
          ↔ (1,365,24,60,60)⊤(1,20,5,3,17) ↔ 33282197.
```

Note that the first element of the vector (1,365,24,60,60) could have been any arbitrary number, since it is discarded in the evaluation of ⊥.

Similarly, (1,3,12)⊥(3,1,6) ↔ 126, the number of inches in 3 yards, 1 foot, 6 inches.

Neither *B* nor *V* need be integral.

```
 24 60 60    763 430.47 110     22563 14545.47 3660
  1  3 12 ⊥ 208 118.97  30 ↔    327   179.07   48
  1 20 12     43  28.07   6      1755   995.07  252
```

The inverse operation *B*⊤*N* is also defined for arrays *B* and *N*.

The result is an array *V* such that *B*⊥*V* ↔ *N*. Hence, ⍴*B*⊤*N* ↔ (⍴*B*),⍴*N* and the "base *B* vectors" run along the first coordinate of the result.

Examples:

```
(5⍴5)⊤2442 ↔ (3,4,2,3,2)

(5⍴¯2) ⊤ 13 ↔ (1,1,1,0,1)

(7⍴2) ⊤ 13 ↔ (0,0,0,1,1,0,1)

(1780,3,12) ⊤ 126 ↔ (3,1,6)
```

This last result is the number of yards, feet and inches in 126 inches.

```
                                                    0 7 2 0
                                                    4 8 3 1
(10,10,10,10) ⊤ (456,7890,2345,123) ↔ 5 9 4 2
                                                    6 0 5 3
```

8.7 Matrix Inverse Operator

Suppose B is a non-singular matrix and $\geq/\rho B \leftrightarrow 1$ so that B has at least as many rows as columns. Then, $\boxplus B \leftrightarrow L$ such that $L+.\times B$ is the identity matrix. L is the left inverse matrix of B and $\rho L \leftrightarrow \phi\rho B$.

8.8 Matrix Division

When you have a vector or matrix A and a matrix B such that $\geq/\rho B \leftrightarrow 1$, then $A\boxplus B$ is defined when $1\uparrow\rho A \leftrightarrow 1\uparrow\rho B$. By definition $A\boxplus B \leftrightarrow (\boxplus B)+.\times A$.

Example:

```
105 72    4  8 5      2 7
 97 56 ⊞  3  9 2  ↔   9 3
114 87    7 10 2      5 4
```

CHAPTER NINE

SUBROUTINES, FUNCTIONS AND OPERATORS

This chapter contains a discussion of how to write subroutines and functions. The difference between the role of functions in APPLE and those in other programming languages is that there is little conceptual difference between functions and operators. Consequently, APPLE can be treated as an extensible language.

9.1 The Distinction Between Functions and Subroutines

A subroutine is a code sequence that is parametrically self-contained. It can be invoked from any part of a program. After the subroutine has completed execution, control is returned to the first executable statement following the point from which it was called.

A subroutine may use or manipulate the contents of variables used by the main program, or it may use or manipulate variables that are accessible only by the subroutine itself. If a subroutine is parametrized, then the values for the parameters are specified at the points of the program at which the

subroutine is called.

A function, on the other hand, always has a value associated with it. Like a subroutine, it may use or manipulate variables used by the main program, or it may use or manipulate variables accessible only by the function itself. A variable can be assigned the value of a function. But, in order for a subroutine to modify variables external to itself, it must explicitly assign values to them.

Subroutines and functions may be anadic, monadic, dyadic or n-adic in that they may take no arguments, one argument, two arguments, or n-arguments, respectively.

If *YORICK* is an anadic function or subroutine, you invoke it by writing *YORICK*. If *POLONIUS* is a monadic function or subroutine, you invoke it by writing *POLONIUS* A, where A is some expression representing the value of its parameter. If *ROSENCRANTZ* is a dyadic function or subroutine, you invoke it by writing A *ROSENCRANTZ* B, where A and B are expressions representing the values of its two parameters. If *GUILDENSTERN* is an n-adic function or subroutine, you invoke it by writing *GUILDENSTERN* ⊂ A;B;C;D;...;Z ⊃, where A,B,...,Z are expressions representing the values of its n parameters.

Note that the monadic and dyadic functions and subroutines are written in precisely the same way monadic and dyadic operators are written. Since functions have values, they can be invoked

from within any expression. Consequently, there is no difference between the use of the functions you define and the APPLE operators. In the remainder of this chapter, we will discuss how you may define and use your own operators in APPLE.

9.2 The Form of a Function or Subroutine Definition

The definition of a function or subroutine consists of a heading, a body, and a footing.

The heading names the function or subroutine, specifies whether it is anadic, monadic, dyadic or n-adic, whether it is a function or a subroutine, and names all of the variables local to its body. The body consists of the APPLE statements that perform the computations and manipulations of the function or subroutine. The footing terminates the definition of the function or subroutine.

9.2.1 The Heading

The heading begins with the symbol ∇. If you are writing a function, you subsequently specify the name by which you will refer to the result in the body of the function, then the specification symbol ←.

If the function or subroutine is to be dyadic, you next list the local name of its left argument.

The following entry is the name of the function or subroutine.

Then, if the function or subroutine is monadic or dyadic, you list the local name of its right argument. If it is n-adic, then you enclose the local names of the parameters in the brackets ⊂ and ⊃, separating the parameters with semicolons.

Finally, if there are to be any variables local to the body of the subroutine or function, you list their names, preceding each one with a semicolon.

Examples: The heading for the anadic subroutine *YORICK* is

```
∇ YORICK
```

This subroutine has no local variables. The heading for the anadic function *HORATIO*, which has two local variables *M* and *N*, is

```
∇ Z ← HORATIO;M;N
```

Here, the value *HORATIO* returns is to be explicitly stored in a local variable called *Z*.

The heading for the monadic function *POLONIUS* is

```
∇ Z ← POLONIUS Y
```

Here, *Y* is the local name of the right argument of *POLONIUS* and the value of *POLONIUS* will be stored in *Z*.

If you wanted to define a monadic subroutine *OPHELIA*, you would write

```
∇ OPHELIA Y
```

OPHELIA was not defined as having any local variables.

A dyadic function *ROSENCRANTZ*, with local variables *Q*,*R*,*S*,*T* would have the following heading

```
∇ Z ← X ROSENCRANTZ Y;Q;R;S;T
```

Here *X* and *Y* are the left and right arguments of *ROSENCRANTZ*, the value of which will be stored in the local variable *Z*.

If *GUILDENSTERN* is to be a 5-adic function of *A*,*B*,*C*,*D*,*E* and has a local variable *N*, you would write the following heading:

```
∇ Z ← GUILDENSTERN ⊂A;B;C;D;E⊃;N
```

9.2.2 The Body

The body is an APPLE program. If the subroutine is a function, the result must be stored into the name you specified someplace in the body of the subroutine and in the heading. It, and any other variable names from the heading may be used as local variables within the body.

The body may also use the names of variables existing outside the body, i.e., they are neither the result, operands, parameters, nor declared local names. Such variables may be modified from within the body.

9.2.3 The Footing

The subroutine of the function definition is terminated with the symbol ∇ on the line following the last line of the body.

9.3 Call by Value

The parameters of a function or subroutine are evaluated from right to left before it is entered. The resulting values are copied into the local storage area of the routine. Then, the body is evaluated.

Therefore, it is impossible to modify a variable existing outside the routine by using the variable as a parameter and then trying to modify it inside the body of the routine. The parameters must be considered as local variables that have been initialized when the routine was called.

9.4 The Scope of Names

Inside a body, you may manipulate global variables defined inside other routines. If several variables have the same name, only one of these is accessible. In order to determine which variables are accessible, you follow the chain of calls backwards. The first time a specific name is encountered, you have found the one that is accessible.

Example: Suppose that *A*,*B* and *C* are names of global variables. There are defined routines *F1*,*F2* and *F3* with the following headings:

```
∇ Z ← F1 A; D
∇      A F2 C
∇ A ← F3 D
```

Suppose that *F1* is called first. Then, inside the body of *F1*, variables with the names *Z*,*A*,*B*,*C* and *D* are accessible. *B* and *C* are global; *Z* and *D* are local, as is *A* since it is the parameter of *F1*.

Assume that *F2* is called from within the body of *F1*. Then, the body of *F2* may operate upon its own variables *A* and *C*, the local variables *Z* and *D*in *F1*, and the global variable *B*.

If the body of *F2* calls *F3*, the body of *F3* may operate upon its own local variables *A* and *C* from *F2*; the local variable *Z* from *F1*; and the global variable *B*.

If the calling sequence was different, quite new effects might appear. For instance, if *F3* is called first, it does not have access to any local variable *Z*. If the body tries to use *Z* in an expression, an error will result. The variable *A* is no longer local to *F1*, but is the global variable by the same name.

9.5 Recursion

Functions may call themselves recursively. In such cases, each

incarnation of the function may be considered as a separate function with its own environment. In particular, the function's local variables are local to it and are global only to those routines it calls non-recursively.

9.6 The RETURN Operator

If you want to exit from a function prior to its completion, you may use the *RETURN* operator. This operator may be used either anadically or monadically.

In the anadic case, *RETURN* returns the current value of the function's return parameter.

In the monadic case, you write *RETURN* followed by a specification expression, which sets the return parameter. For example, you could write:

```
IF A ≠ 0 THEN RETURN Z ← ?A
```

9.7 Comments

Any line in a routine's body can be made into a comment if you start it with the symbol ⍝. The comment symbol is called the "lantern" because it often sheds light on a complicated program.

9.8 Implied Loops

Suppose you define a monadic function *OSRIC*. When you write *OSRIC A*, the function is applied to the array *A* in its entirity.

If you would like to apply *OSRIC* along coordinate *I* of *A*, you write *Z* ← *OSRIC* [*I*] *A*.

Then, *OSRIC* will operate on the hyperplanes along coordinate *I* of *A*, producing a resultant array such that

$$\rho OSRIC\ [I]\ A \leftrightarrow ((I-\underline{I}\underline{O}\underline{R}\underline{G})\uparrow\rho A),(\rho Z),(I+1-\underline{I}\underline{O}\underline{R}\underline{G})\downarrow\rho A$$

where for rows *J* of $\iota(\rho A)[I]$

$$Z[[\underline{I}\underline{O}\underline{R}\underline{G}]J] \leftrightarrow OSRIC\ F\underline{\Delta}A$$

and

$$F \leftrightarrow \lozenge((I=\iota\rho\rho A)/(0;1)),(I=\rho\rho A)/(0;J)$$

CHAPTER TEN

ON THE ORDER OF EVALUATION

The order of evaluation in APPLE expressions is controlled by the rules of right association. However, you can easily construct examples in which it is undesirable to strictly adhere to this philosophy. For example, if we were to write

```
X ← (1,0,1,0,0)/TERRIBLYCOMPLICATEDEXPRESSION
```

we see that *X* will receive only 40 percent of the value that was computed on the right side of the specification symbol. If there is no specification of any part of the remaining 60 percent of *TERRIBLYCOMPLICATEDEXPRESSION* , then we have wasted considerable computing power in its evaluation.

Consequently, there is a slight, but important, modification to the principle of right association in APPLE:

> No portion of a select expression is evaluated unless it is subsequently stored into a variable.

This principle has an interesting consequence. Consider the code:

```
B ←→ (1,0,2,0)
Y ←  (B≠0)\(B≠0)/4÷B
```

Now strict adherance to right association yields disastrous results since the machine must divide 4 by 0 twice. But suppose, instead of producing an interrupt, the result of division by 0 is the undefined scalar □. Then, $4 \div B \leftrightarrow (4,\square,2,\square)$. Continuing the evaluation, we see that $(B \neq 0)/(4,\square,2,\square) \leftrightarrow (4,2)$. At this point, the unwanted □ has disappeared. Next, $(B \neq 0)\backslash(4,2) \leftrightarrow (4,0,2,0)$. Thus, when all has been computed, we have $Y \leftrightarrow (4,0,2,0)$.

This was undoubtedly what the programmer had intended. APPLE's modification to the principle of right association tends to correspond to what programmers find natural.

The APPLE compiler maps all selection expressions into a standard form (given in Section 1.11 of the APPLE specification). The compiler is then able to distinguish the necessary and unnecessary computations, suppressing the latter.

APPENDIX I

AN INDEX OF SYMBOLS

This appendix is an index of APPLE operators. It is divided into two parts: arithmetic operators and array operators.

ARITHMETIC OPERATORS

MONADIC

Symbol	Name	Page
+	identity	18
-	negation	19
×	signum	19
÷	reciprocal	20
*	exponential	20
⍟	natural logarithm	20
⌊	floor	21
⌈	ceiling	21
|	absolute value	21
?	roll	22
~	negation	22
!	factorial	23
○	pi times	23

DYADIC

Symbol	Name	Page
+	addition	25
-	subtraction	25
×	multiplication	26
÷	division	26
*	exponentiation	28
⍟	logarithm	29
⌈	maximum	27
⌊	minimum	27
|	residue	26
?	deal	35
!	combination	34
○	circular function	29
∧	AND	30
∨	OR	31
≠	not equal	31
=	equal	32
⍲	NAND	32
⍱	NOR	32
<	less than	33
≤	less than or equal	33
>	greater than	34
≥	greater than or equal	33

MONADIC

Symbol	Name	Page
⍳	iota	46
⍸	odometer	53
,	ravel	53
⍴	shape	8
[*I*]	partial subscript	56
⍟/	reduction	59
⍟\	accumulation	59
⍤	interval	65
⍉	transpose	73
⌽	reversal	77
/;	subscript generator	55
Δ	whole array	67
⌹	matrix inverse	123
IF		98
ELSE		101
DO		105
UNTIL		108
WHILE		107
LEAVE		109
CYCLE		109
EXIT		109
RETURN		131

DYADIC

Symbol	Name	Page
⍳	index of	113
∈	element of	115
⍷	row of	105
:	laminate	85
,	catenate	64
⍴	reshape	57
/	compression	68
\	expansion	65
Δ	subarray	65
⍙	cross section	67
\(;)	mask	78
/(;)	mesh	79
⍺	prefix	72
⍵	suffix	72
⍉	transposition	87
↑	take	79
↓	drop	76
⌽	rotate	79
←	specification	96
⍒	grade down	115
⍋	grade up	115
∘.⍟	outer product	116
⍟.⍟	inner product	117
⊥	decode	120
⊤	encode	121
⌹	matrix divide	123

APPENDIX II A Fast Fourier Transform

In this appendix a simple program that performs a Fast Fourier Tranform is delineated and described. It is assumed that you are familiar with the way FFT's work, (If not, we recommend that you read "A Guided Tour to the Fast Fourier Transform," by G. D. Bergland, IEEE Spectrum, July, 1969, pp. 41-51. This paper includes a comprehensive bibliography.)

The algorithm we follow is an adaptation of a method developed by R.D. Schmidt and communicated to the author by W. Juran, Proprietary Computer Systems, Inc., Van Nuys, Cal. 91406.

We assume that the input data is an a rank-2 array *B*, where $\rho B \leftrightarrow (L,2)$, with *L* an integral power of 2 not exceeding some fixed number, say, 256. Each row of *B* consists respectively of the real and imaginary parts of a data value.

First, we produce a simple preprocessing function. The algorithms for transformations for time to frequency and frequency to time are essentially the same. The only difference is that in the time to frequency transformation, the result must be divided by the number of data sample points.

With this in mind, we can write our simple driver function, *FAST*. We will make *FAST* dyadic: the second parameter is the data, while the first parameter *A* determines that the transform is time to frequency if $A \leftrightarrow 1$ and frequency to time otherwise.

The code for *FAST* is shown below, where we presume that $IORG \leftrightarrow 1$:

```
∇ Z ← A FAST B;C
[1]    IF ~ ((C ← ⍴⍴B)[2])∈0,⍳8 THEN RETURN Z ← (⍴B)⍴ERR
[2]    Z ← ⍉C⍴FFT((1+2⊗C[2])⍴2)⍴⍉B
[3]    Z[;2] ← -Z[;2]
[4]    ⍝FFT RETURNS CONJUGATE OF RESULT
[5]    IF 1=A THEN Z ← Z÷C[2]
∇
```

Here, we insist in line [1] that the number of rows in *B* be a power of 2 less than or equal to 256. If not, *FAST* returns an array *Z* containing a predefined error value contained in *ERR*.

In line [2], the monadic function *FFT* is invoked, returning an array whose transposed value is stored in *Z*. We will look more closely at line [2], momentarily.

Line [3] produces the complex conjugate of the result of *FFT*.

And line [4] divides by the number of data points if the transform was time to frequency.

When *FFT* is invoked on line [2], its argument is a rank 1+2⊛*C*[2] restructuring of *B* such that ,*B*[[1]1] consists of the pure real data components and ,*B*[[1]2] consists of the pure imaginary data components. For example, if we had started with *B* ↔ ⍉(2,16)(⍳16),-⍳16 then the argument *X* transmitted to *FFT* would have been

X ↔ (2,2,2,2,2)⍴⍉*B*

i.e.,

```
                 1  2
                 3  4
X[1;1;;;] ↔
                 5  6
                 7  8

                 9 10
                11 12
X[1;2;;;] ↔
                13 14
                15 16
```

The imaginary components *X*[2;2;;;] and *X*[2;2;;;] are just the respective negatives of these. The motivation for this restructuring will soon become apparent.

The *FFT* must take care of the required binary sortings on this array. The first such sorting is the one in which the real and imaginary components respectively assume the position determined by reversing the binary encoding of their index in ,*X*[[1]1] and ,*X*[[1[2]]. This can easily be achieved by the transposition, since each coordinate of *X* is indexed by either 1 or 2. The general desired array *Y* is given by *Y* ↔ (1,⌽1+⍳⌊2⊛×/1↓⍴*X*)⍉*X*.

In terms of our example array *X*, this is

```
      Y ←→ (1,5,4,3,2,)⌽X
```

Here,

```
                  1  9
                  5 13
Y[1;1;;;] ←→
                  3 11
                  7 15


                  2 10
                  6 14
Y[1;2;;;] ←→
                  4 12
                  8 16
```

Again, the imaginary component is symetric to the real component.

Next, we need the appropriate array of cosines and sines for the real and imaginary components. These are given by the array *T* where

```
      T ←→ (2,1)∘.○○(ϕ⍳M-1)⌽((M-1)ρ2)ρ0,(⍳¯1+N÷2)×2÷N
```

and

```
      M ←→ ⌊2⊛×/1↓ρX

      N ←→ ×/1↓ρX
```

That is *T*[[1]1] consists of the cosines of the appropriately transposed array of multiples of 2π divided by the number of data points; i.e., 0,(○2÷*N*),(○4÷*N*),...,(○2×¯1+*N*÷2)÷*N*. *T*[[1]2] consists of the sines of the appropriately transposed array of the same multiples of ○2÷*N*.

Before we proceed any further, we list the function *FFT*.

```
      ∇ Y ← FFT X;J;Q;M;N;T
[1]       M←⌊2⊛N←×/1↓⍴X
[2]       Y←(1,⍳M)⌽X
[3]       T←(2,1)∘.○(⍳M-1)⌽((M-1)⍴2)⍴0,(¯1+⍳N÷2)×2÷N
[4]       J←2+⍴⍴T
[5]       WHILE 1
[6]             Q←-/[J←J-1] Y
[7]             Y←(+/[J] Y):[J] (-/[1] T×Q):[1] +/[1] T×⍉[1] Q
[8]             IF J≤1↑⍴Y THEN RETURN
[9]             T←(1,¯1⍳1+⍳M-1)⌽T,[2] T←(1,0)/[2] T
[10]      REPEAT
      ∇
```

Lines [1] - [3] contain the code for defining the initial sorted arrays *Y* and *T*.

The loop spanning lines [5] - [10] iterates until the condition on line [8] is satisfied. The variable *J* is used to control the processing of the data, starting with the last coordinate of *Y* and concluding when the second coordinate has been processed. *Y* and *T* are restructured in lines [7] and [9] to correspond to the sortings required by the algorithm.

The easiest way for you to understand the way the algorithm works is for you to try following its execution on a small array, say one with eight elements. You can then use a form of mathematical induction to verify that it works on the higher-dimensional cases.

Individual differences and group interaction. Since each evaluator will have his own set of values (S_j's), there may be collected diverse values of S_j for the group of evaluators. This divergence can come from differences in R_α values and/or w (weights) values. The Delphi technique or simpler group interaction may be used at this time. The Delphi technique may have been used earlier for both R_α and w values, but the evaluators are not likely to come to a complete agreement on one set of values (but eventually there will be fewer sets than the number of evaluators). Any group interaction may yield some influence toward agreement. Some studies on group planning seem to indicate that people may widely disagree on objectives and criteria but may readily agree to favor certain alternatives.

Let us see how different sets of S_j's for different evaluators can be compared. Let β represent a composite attribute of all α_i's that have been considered. In our park example, β will be "acceptable park design with all the features properly balanced." S_j values can now be used to produce $R_\beta(A_j)$ for each evaluator automatically (associating S_j values with numbers in the interval [0, 1]). A display of R_β's for all alternatives A_1, A_2, ... A_n can be made for each evaluator (see Figure 4). The total display of all such values of all the evaluators may be shown in a scrambled order to maintain anonymity, if desired. In addition, a statistical group response, such as quartiles (Q_1, M (median), Q_3) of each $R_\beta(A_j)$ value, can be calculated. Seeing where his own evaluation stands within the group response may aid him in understanding the overall evaluation.

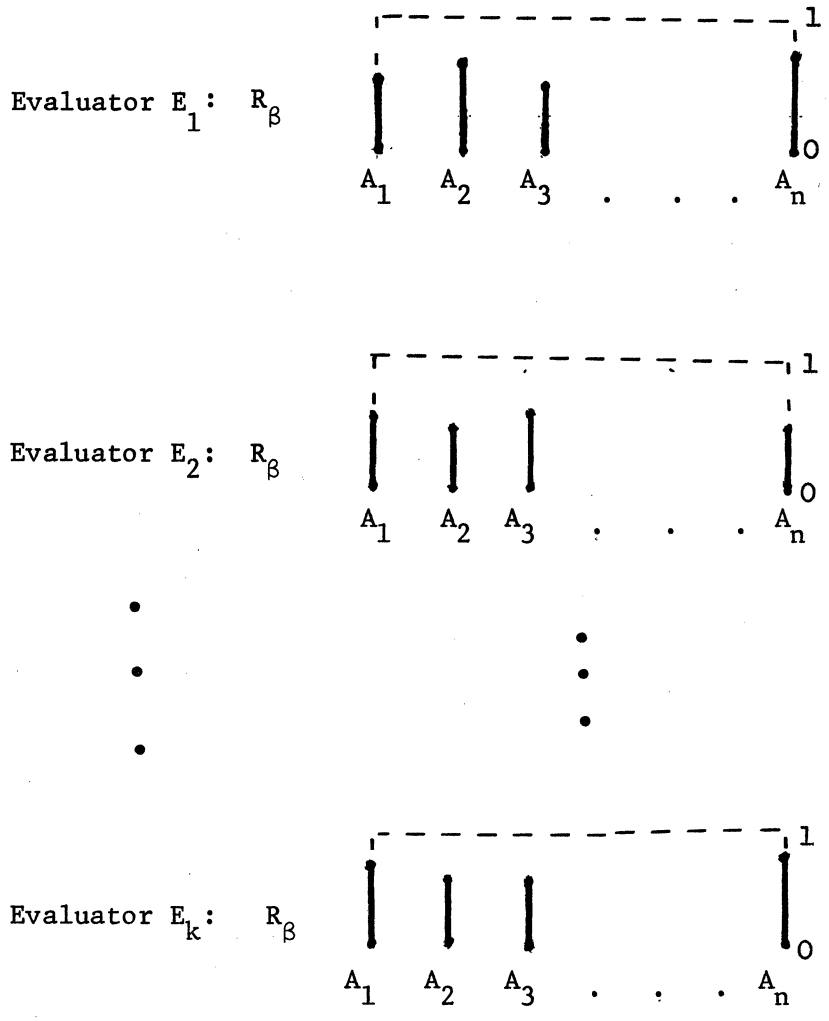

Figure 4. Overall Evaluation of Alternatives

As an evaluator studies the relative merits of alternatives and gains a deeper understanding of trade-off implications, his total conception of the situation grows and his fuzzy-set mapping may become more defined. If $R_{\beta(t)}$ is his preference function operating on the fuzzy-set at time t with respect to the global criterion imposed by β, then as t increases, $R_{\beta(t)}$ tends to converge toward a more precise function; i.e., the evaluator becomes better able to sort out alternatives.

Fuzziness (or impreciseness) of many decision-making situations is usually caused by a mixture of ignorance, randomness, and intrinsic fuzziness. This kind of exercise can help the evaluator to separate out types of fuzziness involved and learn to identify where more information is needed (case of ignorance), where probablistic treatment is needed (randomness), and where increased awareness of other value systems besides his own is needed (intrinsic fuzziness).

In complex decision situations where many competing factors must be properly accounted for simultaneously, the interactive system can be made to keep track of the evaluator's tendencies. For example, suppose the evaluator is excessively cost oriented and his assignments of grades of membership for the cost attributes fall consistently outside the interquartile range (Q_1, Q_3) of the group response. The system can remind him of other important factors and trade-off considerations.

Interacting with other evaluators through the system and seeing where his own evaluation stands within the group response may influence him to take additional factors into account or to adjust his preference functions. If he feels strongly about his preferences, he can try to persuade others by stating the reason why the value should be lower (or higher) than the values (in the interquartile range) expressed by the 75 percent majority.

The process of interaction and reevaluation can be repeated until, it is to be hoped, convergence is attained. Polarization may occur but completely flat distribution is not likely for most value-laden questions. Individuals seem more responsive to value-oriented questions than to factual questions; that is, changes of opinion seem more readily attainable on value-oriented issues than on the factual ones.

SUMMARY STEPS OF MACHINE-AIDED EVALUATION

There are seven evaluation steps to be considered, some of them will be elaborated on in the following pages. For simplicity, the procedure assumes a single user (evaluator) and group interaction is not emphasized. Steps are presented in the "usual" order but can be reordered at the user's direction.

1. List alternatives by name or number assigned to each.
2. List criteria for evaluation in terms of attributes.
3. Rank attributes and assign weights.
4. List values in their "natural" description (numerical or non numeric) for each alternative's attributes.
5. Determine grades of membership of all values of attributes.
6. Calculate the summary value of each alternative.
7. Repeat any or all the steps above.

The list of alternatives and attributes may be prepared in advance covering steps 1, 2, and 4 and can be thought of as an attribute-alternative table (Figure 5). Unlike mathematical tables, this table can contain both numerical and nonnumerical descriptions, even lengthy discussions supplemented by pictures that can be referenced. Therefore, the physical form of the information may not look like the table in Figure 5.

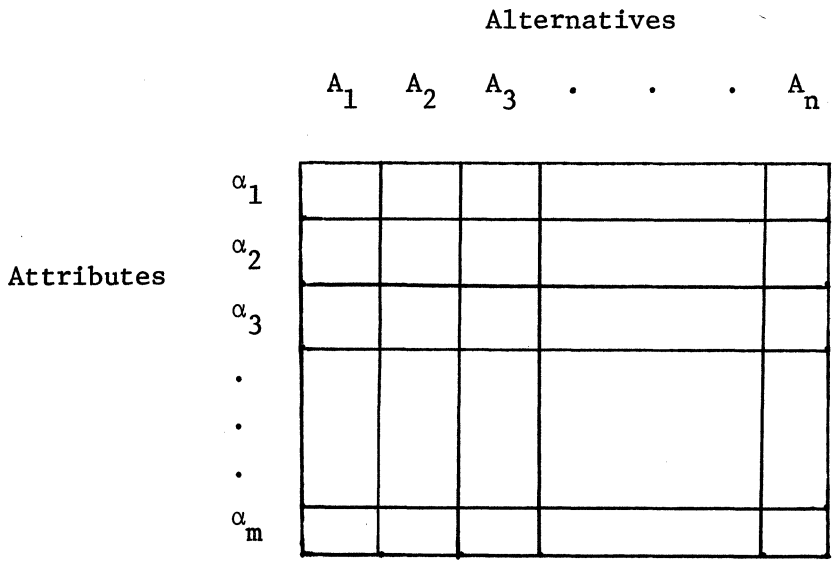

Figure 5. An Attribute-Alternative Table

Following is an expanded description of the seven evaluation steps.

1. List alternatives by name or number assigned to each.

2. List criteria for evaluation in terms of attributes.

Attributes can be given on a noncommittal trial basis with full recognition that they are likely to be inadequate or incomplete; or they can be carefully selected by a group of people (e.g., policy makers, planners, experts, representatives of the public). Attributes may be separated into two groups, "desirable attributes" and "undesirable attributes", or they can be all mixed together. Subsequent instructions will reflect the choice.

3. Rank attributes and assign weights.

Attributes are rank ordered in terms of their relative importance in contributing to the objectives. If there are many attributes and ranking is difficult, machine assistance can be provided by showing only two attributes at a time to the evaluator. Judging the relative importance of two attributes is much easier than ranking the whole list.*

If the evaluator's judgment of importance is transitive and total in ordering, the attributes are listed in the order of importance, possibly placing two or more in the same rank in case of a tie. The logic (computer programs) inside the machine can easily check any inconsistency and ask the evaluator to compare again those attributes whose rankings are in conflict. If the inconsistency is not removed, it is likely that at least one attribute should be redefined in terms of two or more other attributes, or some attributes should be grouped together as one. An experienced evaluator usually can sense which ones are in need of adjustment.

---

* Showing two attributes at a time makes comparison easier but the resulting ordered list should be reexamined as a whole to guard against any possible context shifts, which may result when only two items are compared at a time.

When ranking is complete, the attributes are displayed in rank order. The evaluator is now asked to assign weight w_i to each attribute, starting at the top-ranked one(s) with the weight of 100. Using this as the point of reference, the other attributes are also assigned weights. These weights should reflect the relative "strengths of effects" of attributes contributing to the objectives.

4. List attribute values in their "natural" description (numeric or non-numeric) for each alternative's attributes.

If the attribute values are given in verbose descriptions, they can be condensed to a few key words to be displayed along side the names of attributes. The original information sheets should also be available to the evaluator.

5. Determine "grades of membership" of all the attribute values.

Using the fuzzy-set concept, each attribute value is judged in terms of "grade of membership" -- i.e., a number in the interval [0,1]. Since comparability is important in value judgement, other values of the same attribute from different alternatives can be shown one or two at a time for comparison. In Figure 5, this process corresponds to moving horizontally across the alternative on the same attribute line. When all the attribute values are judged in this way, a new table of values is created within the uniform scale. When it is displayed, it will look exactly like Figure 5 containing a single number (between 0 and 1) in

each cell of the table. Seeing the total array of numbers may prompt him to change his earlier choice of values.

6. Calculate the summary value of each alternative.

The summation $S_j = \sum_{i=1}^{m} w_i R_{\alpha_i}(A_j)$ is calculated for each alternative A_j as one of the basic machine aids, but some other forms of getting the summary value may be tried out. The evaluator can specify his own ideas easily with the man-machine communication language, User Adaptive Language (UAL) (see Hormann, et al [1970]).

If attributes have been separated into "desirable attributes" and "undesirable attributes," S_j's are calculated using only those α's in the desirable category, and $S'_j = \sum_i w_i(1 - R_{\alpha_i}(A_j))$ is calculated using only those α's in the undesirable category. Weights, attached to attributes, remain the same since they should reflect the "strength of effects" regardless of desirability or undesirability. The difference, $S_j - S'_j$ for the alternative A_j may be called the "net-benefit value." Comparing these values presumably will indicate a tentative conclusion concerning which alternatives are best.

7. Repeat any or all the steps above.

The evaluator is encouraged to go back and examine his previous judgments. it is usually advisable, the first time around, to use first impressions in making attribute rankings and in making a judgment of grade of

membership without too much deliberation. Stepping through the whole sequence rather quickly the first time, rather than dwelling on a single factor in detail, will give him a better understanding of how certain factors are accounted for in the total evaluation.

Iterating the evaluation process tends to bring many assumptions into the open, and the evaluator may become more aware of how the conclusions are related to the assumptions. For example, assumptions on the objectives will influence the interpretation of objectives and criteria in terms of attributes and will also influence attribute ranking and weight assignments. Assumptions on political and technical constraints on the proposed designs will certainly influence many decisions. Probing into them with "what if" questions may separate out "real" constraints from imagined ones or those that can be overcome by negotiation or by creative problem solving.

The evaluator may, in the light of new insights and understanding, wish to redefine objectives and specify relevant attributes more carefully. Interacting with the other evaluators, or even with the policy makers, may bring further clarification. Possible use of the Delphi technique has been discussed, so it will not be dealt with here.

Complex trade-off implications, which are typically nebulous, can be made clearer if bar graphs such as those shown in Figure 3 are used. They can be rearranged to show the R_{α_i} values of different alternatives horizontally for each α_i. The evaluator may be encouraged to ask "what if" questions on possible trade-offs that are not evident in the design alternatives; the answers may suggest a new or modified design.

POTENTIAL APPLICATIONS

There are a number of areas of potential application for this method.

1. Complex equipment with many performance criteria.

   Evaluation of different designs of complex equipment such as aircraft and underwater exploration vehicles can use machine-aided evaluation. In these, many attributes must be included in evaluation and they cover both factual information and subjective value information. This class of problems is less fuzzy not only because factual information tends to dominate but because the physical boundary in measuring operational behavior is relatively clear.

2. Selection of suitable locations for large complexes.

   The problem of selecting a suitable location for a large complex, such as a new housing development, often requires careful consideration of many attributes that are qualitative in nature. Among many possible locations, one or a few candidates are usually selected in order to proceed with

designing, legal and financial negotiations, etc. Other examples of large undertakings whose location decisions tend to affect various segments of our society are: manufacturing plants, airports, hospitals, health-care centers, sanitoriums, rehabilitation centers, educational institutions, trade centers, highways, and transportation networks.

3. Complex combination of things that interact.

Making an appropriate EDP system selection from all possible combinations of available hardware/software products to meet the user needs is a complex problem. Guessing at a suitable hardware/software mix is hard enough, but evaluation of a wide number of configurations when the components interact usually requires an advanced modeling technique (Sutherland [1971]). Information on the performance characteristics of hardware and software components are separately available, but very little information can be had on the total performance characteristics for specific configurations--unless the pieces are all of the same manufacture. After modeling produces the system's performance characteristics, our technique can be used in total performance evaluation.

A similar situation facing the decision maker is the selection of alternative designs of hospitals, schools, housing complexes, or research laboratories.

Another area of interest is compensation programs to provide employees many different options. Companies who can provide many options will have a definite advantage in employee inducement and retention.

4. Long-range large scale programs comprised of many projects that are interrelated and interdependent.

Many government programs such as health care, welfare, education, and foreign aid programs are in this category. This is an area of great importance because of its far-reaching effects, both intended and unintended. It is also the area of greatest difficulty because of its complexity, unclear boundary (sphere of effects are not clearly definable) and future-oriented consideration.

These programs or measures that tend to create many side effects or that produce long-term effects or irreversible conditions, must receive extra care in planning. Although the future is always uncertain and, therefore, nc forecasting techniques can claim total accuracy, a variety of forecasting techniques combined with modeling can produce some indication of types of impacts a given program might make in the future.

After possible consequences of alternative courses of action are generated, the consequences can be arranged within a "decision-event map," indicating interrelation of actions taken, their intended results and possible side effects, and intervening events that are likely to happen.

Concentrating on the consequences in the time-stream (rather than at one point in time), our technique can still be employed by using attributes that explicitly indicate future impacts. (e.g., "rate of yearly increase in food production in country X, during 1970-1975, after introduction of farming equipment" or "number of farm workers in country X migrating yearly into cities during 1970-1975").

Admittedly, any future-oriented evaluation is very tenuous. However, evaluating programs to assist underdeveloped countries is a more amenable problem than evaluating our own future possibilities. We can use the U.S. and other developed countries as models in planning to avoid possible undesirable consequences and to promote those attributes that are desired by the country. Although exact correspondence between the model and the real consequences in a given country cannot be expected, hindsight is readily available while foresight is not.

SUMMARY

The importance of including many criteria of various types and degrees of imprecision has been discussed. The man-machine fuzzy-set approach described here is our first attempt to tackle this task. Insight gained in using these techniques may lead to improvements or to new ideas and techniques.

Systematic analyses of the situation supplemented by intuitive judgment was emphasized. The following points may be worth reviewing:

- Consistency in treatment of all alternatives with many attributes describing desirability or undesirability. One aspect of consistency achieved here is the making of everything into a value-oriented judgment; even though attribute values may be factual, determining their worth in relations to the objectives requires a judgmental decision. The fuzzy-set concept allows explicit treatment of imprecise value judgments.

- Comparability. Since absolute judgment is far more difficult than relative judgment, the man-machine techniques facilitate comparison by bringing in other relevant factors. In addition, the fuzzy-set treatment of attribute values make them commensurable, and complex trade-off possibilities can be explored much more readily than without such assistance.

- Systematic use of the knowledge and experience of experts as well as opinions of people from different backgrounds. Those techniques (such as the Delphi) for direct involvement of people can fit naturally into the on-line interactive system.

REFERENCES

Dalkey, Norman. "An Experimental Study of Group Opinion: The Delphi Method," FUTURES, Vol. 1, No. 5 (1969), pp. 408-426.

Dalkey, Norman. "Analyses from a Group Opinion Study," FUTURES, Vol. 1, No. 6 (1969), pp. 541-551.

Helmer, Olaf. "The Delphi Technique and Educational Innovation," in O. Helmer et al, Social Technology, New York: Basic Books, Inc., 1966.

Hormann, Aiko M., David Crandell, and Antonio Leal. User Adaptive Language (UAL): A Step Toward Man-Machine Synergism. SDC document TM-4539. April 1970.

Hormann, Aiko M. "A Man-Machine Synergistic Approach to Planning and Creative Problem Solving: Part I," International Journal of Man-Machine Studies, Vol. 3, No. 2 (1971).

Hormann, Aiko M. "A Man-Machine Synergistic Approach to Planning and Creative Problem Solving: Part II," International Journal of Man-Machine Studies, Vol. 3, No. 3 (1971).

Kamnitzer, Peter, and Stan Hoffman. "INTUVAL: An Interactive Computer Graphic Aid for Design and Decision Making in Urban Planning," Proc. Second Annual Environment Design Research Assoc. Conf., (1970) pp. 383-390.

Sutherland, John. "The Configurator: Today and Tomorrow," Computer Decisions, February 1971, pp. 38-43.

Zadeh, L. A. "Fuzzy Sets," Information and Control, Vol. 8 (1965), pp. 338-353.

SYSTEM DEVELOPMENT CORPORATION ▪ 2500 COLORADO AVENUE ▪ SANTA MONICA, CALIFORNIA 90406 ▪ (213) 393-9411