HARVEY BINGHAM

A GENERALIZATION OF *APL*

JAMES ARTHUR BROWN

SYSTEMS AND INFORMATION SCIENCE
SYRACUSE UNIVERSITY

A GENERALIZATION OF *APL*

by

JAMES ARTHUR BROWN

B.A., Gannon College, 1965
M.S., Syracuse University, 1969

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Systems and Information Science in the Graduate School of Syracuse University, September, 1971

Approved_____

Date_____

Copyright 1971

JAMES ARTHUR BROWN

ERRATA

The following correct known technical errors but not minor typographical errors.

pg 40 after line 12 add the line:

ELSE undefined

pg 40 line 16

for each scalar integer *I*, *ORG*≤*I*<*ORG*+ρ*L*

not *ORG*≤*I*<*ORG*+ρ*R*

pg 45 line 7

integer scalar instead of numeric

pg 45 line 14 substitute + for -

IF ≥ -ρ,*R* *THEN* ((⍳|*L*)+(ρ,*R*) ⌷ *R*

rather than .˙ (ρ,*R*) - *L*) ...

pg 52 line 7 add

element vector. A scalar is extended to the dimension of a vector operand.

pg 57 line 6 should read:

Z ← → ((¯1+*ORG* + ρ*R*) + *ORG* - ⍳ ρ*R*) ⌷ *R*

rather than *Z* ← → ((¯1+*ORG* + ρ*R*) -⍳ρ*R*) ⌷ *R*

pg 58 line 6 should read

Formal description: 0 origin dependent

pg 80 line 6 should read:

Formal description: 0 origin dependent

ERRATA (continued)

pg 103 line 7 should read

set of rank *N-M* arrays joined along *M* new coordinates.

pg 121 line 4 should read

Formal description: 0 origin independent

pg 121 line 5 should read

$\underline{Z} \leftarrow \rightarrow \supset[\underline{ORG}]\ ([\underline{ORG}]\ \underline{L}) \circ . \S \top \underline{R}$

pg 126 last line should read

$^{-}2\ ^{-}2$

pg 206 line 6 insert parens to read

$\perp X \S (\nabla' Z')\ \nabla' Y'$

pg 209 line 22 substitute + for - to read

$[6]\ (\rightarrow 0)\ Z \leftarrow ((\iota|L) + (\rho,R) + L) \S R$

This list does not contain errors occurring in the text which do not alter the meaning of a passage.

PREFACE

This paper presents a generalization of *APL*. It is presented as a notation not as a computer implemented programming language although implementation considerations are eluded to. The fact that the body of the work contains 2*8 pages is not an implementation consideration.

For completeness some of the sections present *APL* functions or concepts unchanged from *APL\360* (which is used as the standard for comparison). Parts of many pages are blank for two reasons. First, starting new sections and function descriptions on a new page make reading and reference easier. Second, this work was prepared entirely in *APL* using a text editor.

I would like to acknowledge the following people who read selected chapters of this paper and provided valuable suggestions. Ted Edwards, Control Data Canada; Bill Jones, Syracuse University; Dick Lathwell, IBM Philadelphia Scientific Center; and Bill Newman, Syracuse University. Special thanks go to my advisor Dr. Garth Foster who struggled diligently through drafts containing illegible definitions and muddled examples yet still managed to prompt some of the more significant discoveries. Finally I want to thank my wife Karen who prepared the index to this paper not to mention many midnight meals.

CONTENTS

Chapter 0

Introduction

A. Notation

One of the best measures of a civilization is its developement of notation systems for expressing ideas in writing. Most common is the written form of spoken languages. But these notations are subject to ambiguity, idiom, and imprecision.

When an idea must be precisely written, the notation used tends to become symbolic in that special graphics are assigned well defined meanings. One problem with the modern sciences is that each branch uses its own set of symbols and worse each uses the same symbol to stand for different functions. Even the symbol + is not sacred and is often used for the logical OR predicate when arithmetic is not also a requirement. In algebra the same symbol may be used over and over again for different functions. For example, × is used to denote the product function in different groups. This is not bad if the functions themselves are the objects of study and it shows that the meanings assigned to symbols are indeed arbitrary and may be changed if desired. However if specialists are to communicate with each other with ease, a common notation is needed. This problem was underscored by the advent of the general purpose computer. A notation common to the computer and to the users of the computer was

required. The first attempt to solve this problem was to invent new notations called programming languages which mimicked existing notations (e.g. arithmetic). Thus the rule "multiplication before addition" was incorporated in almost all programming languages (and why not? - it can be convenient and everybody knows of it). When new functions were included (like exponentiation) they were given a position in the hierarchy. A simple Fortran compiler had the following hierarchies:

assignment	lowest
addition and subtraction	
multiplication and division	
exponentiation	
everything else	highest

where everything else included user written subroutines [Leeson 13]. This isn't so bad and is easy to remember. But if more functions are included (logarithm, relationals, etc.) the table becomes large and bothersome to remember.

Many of the rules used in mathematics are artificial and are only for convience. When they cease to be a convenience and become a nuisance, then perhaps they should be put aside. A programming language ignoring these rules would involve more forgetting than learning.

B. *APL* Breaks the Rules: History

APL is a notation created by K. E. Iverson and presented to the public in 1962 in his book "A Programming Language" [Iverson 10], from which the shorthand name arises. (some claim that the name *APL* has its roots in the Greek word απλα meaning simple [McDonnell 15]. Everyone knows that an APLITE is a simple rock.) This notation has cast out the old precedence rules of mathematics allowing functional richness to coexist with simplicity of evaluation. Its integrated use of arrays permits subjugation of unnecessary detail. Its descriptive power was shown in 1964 when "A Formal Description of System/360" appeared in the IBM Systems Journal [Falkoff et.al. 7] with the actual description taking under 20 pages.

Then 1966 saw the advent of an experimental *APL* time sharing system and soon the habits of people at IBM's Yorktown Heights began to include *APL* as a research tool. This experimental system evolved into the present *APL*\360 which is now a program product of the International Business Machines Corporation [IBM 9].

APL has become an object of study and implementation by computer manufactures and universities. Interest in *APL* continues to grow and the question may be asked: "Where do we go from here?"

C. Where do we go from here: The Problem

APL is one of the most powerful notations for computer programming in existence today. It is simple to learn because its rules are few and simple. It is amenable to mathematical analysis because it is self-consistent.

Yet the notation clearly indicates directions for generalization which will simplify or complete existing concepts. Other extensions are prompted by particular applications of the notation.

Many areas of study use data structures not easily represented by the simple rectangular arrays of *APL*. For example the study of formal systems often gives rise to tree structures.

The power of the notation is not available for function definition or modification. Expressions may be written but only the values and not the expressions themselves may be manipulated. It is desirable that one function be able to treat another function as data. It should be possible to implement function definition as an *APL* function. It should be possible to implement other language to *APL* compilers as *APL* functions. These desires become requirements when one postulates *APL* as the native language for a machine.

The purpose of this paper shall be to discuss these problems and to propose solutions for them.

D. Where do we go from here: The Solution

This paper presents a generalized *APL* notation. While a knowledge of *APL*\360 is not a prerequsite for reading this paper, it is required to distinguish between what currently exists in *APL* and what is being proposed. A concise presentation of *APL* may be found in "*APL*\360 Reference Manual" [Pakin 18]. A more recent publication designed for self-teaching is "*APL*\360 An Interactive Approach" [Gilman,Rose 8] . This volume contains a supplement describing the most recent improvements to *APL*\360.

Appendix 1 to this paper summarizes the changes to *APL*\360 which are proposed. The generalizations fall into four classifications:

1.) Syntax
2.) Arrays
3.) Names
4.) Functions

Generalized Syntax

The syntax of *APL*\360 may be characterized as follows:

a.) Precedence is positional - functions within an expression are evaluated from right to left

subject to parentheses.

b.) Context sensitive - a function defined within the notation (a primitive function) may be used for two different functions depending upon the existence of two operands (with infixed function symbol) or one operand (with function symbol on the left). A function described by a set of expressions (a defined function) may have zero operands.

The generalizations proposed here involve first a closer tie between primitive and defined functions so that the notation becomes functionally extensible. The notation is in no way made syntacticaly extensible. A function symbol used to name primitive functions or an identifier used to name defined functions may be used to denote any or all of four functions depending upon the existence of zero, one on the right, one on the left, or two operands. To do this requires only a small modification to the present parentheses rules. The second generalization of syntax permits functions having no result to exist within an expression. The bracket notation for array indexing is deleted and replaced by a more selective dyadic function for indexing. Lastly, a defined function may have a function index.

These generalizations simplify the syntax of the

notation by uniformly treating primitive functions and defined functions as instances of the same class of objects.

Generalized Arrays

The arrays of *APL\360* are simple rectangular arrangements of scalar arrays (i.e. values of the array). Scalar arrays may be one of two types: numeric or character. While arrays of mixed type are not permitted in *APL\360*, they are well defined. Therefore the inclusion of such arrays, as done here, is not really an extension.

The first generalizations to arrays proposed are the definition of new scalar types. Most notable is the program scalar which permits expressions and functions to be treated as data. The inclusion of this scalar (along with one function and one operator) implies a significant broadening of the capabilities of the notation. It permits functions to define and modify other functions and solves the other problems mentioned earlier, but it has unexpected benifits as well. A "call by name" facility is achieved by passing program scalars as operands to defined functions. Multiprogramming of *APL* functions is an immediate consequence of arrays of program scalars and is achieved by exploiting the already existing parallelism of the notation.

The second generalization is the obvious extension from arrays of scalar arrays to arrays of arbitrary arrays. The properties of such arrays really arise from the functions

defined on them and it is the functions which are of primary interest.

The Name Domain

Names in *APL\360* stand for either functions or variables (or other extra-lingual objects not considered in this paper). Mention of a variable name usually implies a reference to its associated array. The exception is a name occuring as the left operand of a specification. Mention of a function name implies evaluation of the expressions comprising the function definition.

The generalizations proposed in this paper make functions and variables sub-classes of the same set of objects (appropriately called variable functions). A defined function is a name associated with a scalar program array. Mention of any name usually implies evaluation of its associated array. In this case the name-array pair is said to exist in the value domain. Again the exception is a name occuring as the left operand of a specification. In this case the name is said to exist in the name domain and it is not in general important that any array be associated with the name. Expressions are permitted on the left of specification and they evaluate to arrays of name scalars. Functions occuring in such expressions must be defined for name arrays and generally mimic functions in the value domain. Using the same function symbols in both domains

makes the new functions easy to remember but increases the sensitivity of expressions to their context.

These concepts already exist in *APL\360* but are hidden by the use of the special bracket notation used for array indexing.

Generalized Functions

Some functions defined in *APL\360* have been trivially extended to include the new data types in their domain. Some functions previuosly defined on scalars or vectors are extended to include higher rank arrays. Functions are defined to create and manipulate general arrays.

A general array extension method is developed which uses definition of a function on vectors to extend the function to array operands. This method makes definition of functions on arrays easy and more importantly may itself be used to extend functions to arrays in non-standard ways. The existence of this method motivates the definition of functions only on vectors in the early parts of Chapter 1. A small number of new functions have been defined to permit special action on the new data types.

E. Guidelines

The determination of just what constitutes good *APL* or what extensions or generaliztions are in the *APL* spirit is largely a matter of taste. The following guidelines are followed in this paper:

1.) Existing definitions are preserved whenever possible - extend don't change.

2.) The useful identities are preserved whenever possible. Identities are used to infer the properties of general arrays.

3.) The introduction of special syntax is avoided - keep it simple. The temptation to include new heirarchies of punctuation (i.e. braces, bars, vinvulae, etc..) is resisted. The use of the semicolon (;) as a low precedence spearator is expunged.

Chapter 1

Primitive Functions and the Value Domain

A. Introduction

In this chapter the computational subset of the notation is developed. The basic data objects are defined and their properties are exploited by the functions defined on them. When the significant feature of an operand to a function is its value (the data object which it defines) then the operand is said to be in the value domain. The majority of the functions described in this chapter already exist in *APL* and are trivially extended. Notable exceptions are the functions to create and manipulate general arrays; the scalar product operator which applies functions uniformly to sub-levels of general arrays; and the general array extension method for defining vector functions on array operands.

B. The Meta Language

The following meta-notation is used throughout this paper in verbal descriptions and formal definitions. A certain basic knowledge is assumed here. It is presumed that the meanings of the terms function, operand, etc. are already known even though some of them are formally defined later. Following the list of meta notations are some simple examples which justify the assumption.

D - a dyadic function (one with two operands).

M - a monadic function (one with one operand on the right).

L - the left operand of a function.

R - the right operand of a function.

Z - the result of applying a function to its operand (i.e. the result produced when the function is evaluated).

E - an empty vector.

S - any scalar.

U - any unit array.

V - any vector.

□ - an array of 2 dimensions.

FI - a function index.

ORG - the index origin.

PROD X - a scalar which is the product of all the elements in the numeric array *X* (=1 if *X* has no elements).

↔ - identically equal (the objects pointed to are the same objects). the symbol ↔ may be read "is".

↔/→ - not identically equal (the objects pointed to are different). the symbol ↔/→ may be read "is not".

IF X THEN Y ELSE Z - this is the expression of the conditional and has the value *Y* if *X* is true (*X*↔1) and the value *Z* if *X* is false (*X*↔0) and is otherwise meaningless. If the *ELSE* clause is omitted or "*ELSE* undefined" is written then the value is meaningless in the case *X* is false. *X* is called the antecedent, *Y* is called the consequent, and *Z* is called the alternative.

SCALAR X - this predicate is true if *X* is a scalar and false otherwise.

X AND Y - this is the expression of the conjunctive and is defined in terms of the conditional.

```
X AND Y ↔
IF X THEN
   IF Y THEN true (i.e. 1)
      ELSE false (i.e. 0)
ELSE IF Y THEN  false
     ELSE false
```

The last statement is included to make *AND* meaningless in the case *X* false *Y* neither true nor false.

Example of a function and operands:

2+3

In this case the function + is recognized to be that of addition. *L* is 2, *R* is 3 and *Z* is 5. Thus the terms function and operand are merely labels for familiar objects. *Z* of this example could be expressed using the conditional as:

```
Z ↔
IF L ↔ 3 THEN
   IF R ↔ 4 THEN 7
```

or

```
Z ↔
IF (L ↔ 3 AND R↔4) THEN 7
```

Since *AND* is defined in terms of the conditional it is not strictly needed. However the above formulation shows that it can be used to limit the depth of recursion of the conditional in cases where the recursion is not of interest.

Some of the meta-notation is parallel to *APL* functions (i.e. the meta-notation "identically equal" and the *APL* function "same" yet to be introduced). Many functions could be made pure *APL* by substituting functions for meta-notation. The meta-notation is used instead of functions for the following reasons:

1.) It is a convenient starting point for defining functions and structures.

2.) Unnecessary circularity of definitions is avoided.

3.) Some parenthesis are avoided without loss of clarity and with an increase of readability.

4.) It focuses attention on the object being defined.

5.) The formal definition of a meta is never a point to ponder while it is with a function.

C. Constant Functions, Arrays

The first class of functions to be introduced are called constant functions. A constant function (or an array) is a function whose name and value are intimately associated in that the value may be determined solely from the name. These functions are the basic building materials for all other functions.

Literal Functions

A literal function (or literal array, literal) is a constant function whose constant value is precisely and immediately determined by its manifestation. The simplest form of literal function is a scalar. A scalar is an undefined object which may be described as having a single value but having no coordinates (i.e. no direction, no structure, empty dimension).

A scalar may be thought of geometrically as a point. Following are scalars of five distinct types three of which have literal manifestations representing the values of the scalars.

Character scalar

'A' the single character *A*. Enclosing quotes are used so there can be no confusion between the character '3' and the number

3. The quote character itself is represented by two quotes in the usual manner.

Numeric scalar

35 the single number thirty five.
35 is scalar because it is the number which is of interest and not its multidigit decimal representation.

Position scalar

⍬ The position scalar (hereafter called the p-scalar). This scalar is used as a placeholder when no other type is appropriate.

Program scalar

Every piece of notation in *APL* is a program scalar. Its mention implies its evaluation thus the three scalars mentioned above are program scalars. However '*A*' evaluates to a character scalar, 35 evaluates to a numeric scalar, and ⍬ evaluates to the position scalar.

Name scalar

This scalar refers indirectly to a value.

Notice that while all values are printed as characters,

character type values are still distinguished from numeric type values by use of enclosing quotes. Since it is useful to supress quotes on display of a function result, confusion may still exist when reading such a result (but never in evaluation). A simple array is an ordered arrangement of zero or more scalars along zero or more coordinates. The number of coordinates in an array is called its rank. A scalar is then a 0 rank simple array. An array of rank 1 is called a vector. An array of rank 2 is called a matrix. The dimension of a simple array is a simple vector telling the number of scalars along each of the coordinates of the array. If an array has zero scalars along any of its coordinates (i.e. if a zero occurs in its dimension vector) then it is called an empty array. In particular the dimension of a scalar is the empty vector (zero scalars along one coordinate). The scalars used in an array are collectively called the values of the array.

Examples:

5 ¯3 is a rank 1 array (vector) of dimension 2 containing the scalars 5 and ¯3.

0	1
2	3
4	5

is a rank 2 array (matrix) of dimension 3 2 containing the scalars 0,1,2,3,4,and 5. (selecting the values of an array one row

at a time as above is called selecting them in row major order).

'ABCD' is a rank 1 (vector) of dimension 4 containing the scalars *'A'*,*'B'*,*'C'*, and*'D'*.

A literal array may be considered the written form for the value of the array.

D. Display of Simple Arrays

If the printed form of an array is to be an unambiguous representation of the array, then the dimension of the array and the scalar value at each position in the array must be clearly indicated. Therefore the following display format is used:

1.) Dimension display - the symbol ρ followed by a vector of integers which is the dimension of the array.

2.) Value display - A scalar is displayed as its literal value. A vector is displayed as a line consisting of its scalar values. A matrix is displayed one row per line. Each plane of a higher rank array is displayed as a matrix with one blank line separating planes, two blank lines separating hyper-planes, etc.. In this paper all printing is double spaced. This requires some visual compensation.

For many arrays, the dimension display is a repetition of information contained implicitly in the value display. In these cases the dimension display may be elided without loss of information. A scalar may always be displayed without a dimension display. A one element vector requires the

dimension display to differentiate it from a scalar, while a two element vector does not require it. In general, if the dimension neither contains a zero nor begins with a one, then it may be elided.

In an implementation it may be convenient to always elide the dimension display, but when defining functions as will be done here, it is important that no ambiguity be present.

E. Primitive Functions and Simple Expressions

The second class of constant functions are expressions. Since expressions are usually involved with the class of variable functions called primitive functions, they shall be discussed together.

A primitive function is a function which is composed of a name and an associated scalar array of type program. The name of a primitive function is a special symbol (i.e. +,-,×,÷,etc.) which is called a function symbol. A primitive function is not a constant function because its name does not indicate its description. In general the value of a primitive function depends upon the values of other arrays which are called operands of the function. A function having two operands is called dyadic. A function having one operand is called monadic or dextri-monadic [McDonnell 15]. A function having zero operands is called niladic. Using these classifications all the arrays previously introduced (i.e. literal functions) may be considered niladic functions because their value depends upon zero other arrays. A function taken with some particular operands is called a simple expression or just an expression. The process of determining the value of an expression is called evaluation of the expression and is considered a mapping of the operands of the function into the result. This is an important distinction. A function (say addition +) is a

mapping and to properly define a function it is only necessary to specify the mapping. An expression (say 3+4) specifies an application of the mapping to the particular operands. Thus an expression is an alternate notation for some constant (3+4 is an alternate notation for the constant 7).

The Syntax (form) of Primitive Functions

dyadic *L*÷*R*

The function symbol '÷' appears between the left and right operands *L* and *R*

monadic ÷*R*

The function symbol appears to the left of its single operand *R*. There is no confusion with the dyadic function which uses the same function symbol because there is no left operand.

dextri-monadic *L*÷

The function symbol appears to the right of its single operand *L* (hence the term dextri-)

niladic ÷

There are no operands.

Thus the number and positions of the operands of a function uniquely and unambiguously determine which of the four possible functions a function symbol stands for. The description associated with the function name must be

sensitive to each environment in which use of the function is to be valid. It will be seen that an effort is made to define functions having the same name but differing valid syntax, related in meaning so they will be easy to remember. However it should be noted that this is an attempt at good taste and not a requirement. In these examples it is the syntax of the functions which is of interest. The meanings of *L*÷*R* and ÷*R* will shortly be presented. The meanings of *R*÷ and ÷ will remain undefined. Dextri-monadic and niladic functions are of little importance till defined functions are introduced but are in fact valid forms for future expansion of the language.

F. Scalar Functions

A scalar function is a function which is defined for scalar operands and which produces a scalar result. Appendix 2 summarizes the primitive scalar functions and gives their identity elements. They will not be discussed individually except by example.

Examples

```
      ÷5
.2
      10÷4
2.5
```

Scalar functions are extended to take array operands according to the following rules:

For monadic scalar functions (of either type):

> 1m) If the operand is a non-scalar array then the function is applied to each scalar in the array producing a result having the same dimension as the operand.

For dyadic scalar functions:

> 1d) If the operands are non-scalar arrays then they must have the same dimension and the function is applied to scalars in corresponding positions in the two arrays producing a result having the same dimension as the operands.
>
> 2d) If one operand is scalar and the other is a non-scalar array, then the function is applied to the scalar operand paired with each scalar of the non-scalar array producing a result having the same dimension as the non-scalar operand.

These rules are often called the scalar extension and they will be formalized after the proper theory has been

developed. In the following examples arrays of rank 2 are enclosed in a box (i.e. meta □) because there is so far no linear representation for such an array.

Example scalar functions:

```
      ¯4  5  6.5
¯4  ¯5  ¯6.5
```

```
      4 0 ¯4 | 7 7 7
3 7 ¯1
```

```
      ⌈ ┌──────┬──────┐
        │ 3.14 │  .7  │
        ├──────┼──────┤
        │  4   │ ¯3.14│
        └──────┴──────┘          (i.e. a 2 2 array)
4  1
4 ¯3
```

```
      ┌──────┬──────┐ ⌈ ┌──────┬──────┐
      │ 3.2  │  2   │   │ 3.4  │  2   │
      ├──────┼──────┤   ├──────┼──────┤
      │ ¯.7  │ 1.5  │   │ ¯2   │ ¯.6  │
      └──────┴──────┘   └──────┴──────┘
3.4  2
¯.7  1.5
```

```
      E ⊛ E        (E ↔ empty vector)
E
```

```
      0 > ┌──────┬──────┐
          │ 3.14 │  .7  │
          │ 4    │ ¯3.14│
          └──────┴──────┘
0  0

0  1


        ¯3  0  3 | 5

¯1  5  2


       E×7

E
```

Note that in each case the result has the same dimension as the non-scalar operand.

G. Compound Functions and Compound Expressions

The *APL* language does not provide symbols for every possible function. In particular no functions of three or more operands are permitted by the syntax. The functions which are provided (i.e. the primitive functions) are those of general usefulness which can be used to define more specialized functions. Such a composite function is called a compound function and is composed from zero or more primitive functions. As an example let A,B, and C stand for any three arrays. Then a candidate for a compound function is :

$$A\times B+C$$

since whatever it is, it is defined in terms of primitive functions. It has been seen that given a primitive function and its operands, it is never a problem to evaluate the function (i.e. just apply the mapping). But here it is not clear exactly what the operands of the primitive functions are. This ambiguity is resolved by introducting punctuation which describes the function-operand relationship. The symbols used for the punctuation are (and) and they surround a function and its operands. Thus

neither (case 1) $((A\times B)+C)$

nor (case 2) $(A\times(B+C))$

contain any operand ambiguity. Unfortunately this rule implies that the expression 3+4 must be written (3+4).

And in fact a pair of parentheses is introduced for each primitive function used. While this is theoretically no problem, it is bothersome for people to write or look at so many parentheses. One way to limit them is to assign precedence to functions to indicate which should be evaluated first. Mathematics and traditional programming languages do this. But there are too many primitives in *APL* to make this workable so instead the following two parentheses elimination rules are adopted:

1.) Parentheses which delimit the right operand of some function may be deleted.
2.) Parentheses which do not alter the scope of operands for some function may be deleted. (i.e. at least the outermost pair)

These rules impose the following precedence on functions:

> If a function occurs immediately to the right of a right parenthesis, then it has lower precedence than any function occuring in the parenthesized expression. Otherwise precedence is positional and increases from left to right [Lathwell,Mezei 11].

Using the parentheses elimination rules the previous example may now be written:

(case 1) $(A \times B)+C$

and (case 2) $A \times B+C$

Note that a compound function is still a function and is considered a mapping of its operands into its result. When a compound function is written with explicit operands (i.e. $2 \times 3+1$) it is called a compound expression and as before is a constant. Evaluation of all such functions can be considered a table look-up procedure. Thus there is no fundamental concept of one primitive of a compound function being evaluated before another (i.e. no time relationship). The many function symbols are just an unusual syntax for the specialized function. However since it is not convenient to remember (or compute!) the tables for all possible functions, people and computers use algorithms to evaluate compound functions and it is these algorithms which suggest a time relationship.

Such an algorithm for evaluating compound expressions using the above two rules may now be stated:

> Evaluate functions in order of decreasing precedence.

This statement is actually stronger than required since in $(A \times B)+C \times D$ it does not matter which multiplication is done first. The algorithm is sometimes called by the misleading name "the right to left rule". These rules are

sufficient to evaluate any expression. The section on magic syntax (Chapter 3) discusses this more and shows that the order of evaluation described in A Formal Description of *APL* [Lathwell,Mezei 11] does not hold with the generalized syntax.

Examples of compound expressions:

```
      3×4+1
15

      (3×4)+1
13

      -4+1
¯5

      (-4)+1
¯3

      0⌈⌊1+10●139
3
```

H. Function Presentation

The section following this one begins the exposition of non-scalar functions. Each function is coded with a letter to indicate the degree to which the function differs from existing *APL* functions. The codes have the following meanings:

I. inclusion - a function existing in *APL* which is retained unchanged or trivially extended to include new data types or general arrays.

E. extension - a function existing in *APL* which is generalized or altered.

A. addition - a function not currenetly defined in *APL*

The functions are generally presented in the following format

1.) syntax - a display of the function symbol to be used for the function and the number and positions of its operands.

2.) english description - a brief sentence indicating the task performed by the function.

3.) conformability - general information about what data objects are in the domain of the function.

4.) formal description - defines the action of the function.

A formal description is in order whenever the function being described can be defined in terms of the meta-notation and previously defined functions.

Some functions in the notation have hidden or implied operands as well as those explicitly written. Of chief interest is the index origin.

Particular elements of an array are sometimes refered to by a set of one or more integers called the index of the element. For example the elements of a vector would be refered to by succesive integers. Any integer may be chosen as the index to the first element of a vector. This integer is called the index origin and is denoted *QRG* in the meta-notation. The second element of a vector is always refered to by *QRG*+1. An element of a rank *N* array may be refered to by a set of *N* integers. While one could write defined functions which used *N* origins for a rank *N* array, primitive functions always use the same origin for all coordinates. The important consideration is that functions are sensitive to the index origin and arrays are not. An

array is an arrangement of objects and is independent of how one decides to number the objects.

When a formal description of one function includes previously defined functions then it is possible that the description is origin sensitive. Therefore each description is labeled as follows:

1.) origin free - the index origin does not affect the description.

2.) origin independent - the index origin does affect the description and is explicitly mentioned via the meta $\underline{ORG}$.

3.) 0-origin dependent - the index origin does affect the description but $\underline{ORG} \leftrightarrow 0$ is assumed.

Formal descriptions have the following properties:

1.) They are recursive - evaluation of the conditionals involves the function being described.

2.) recursion is finite - there is at least one consequent or alternative which does not involve recursion and all operands declared conformable lead to these.

3.) they are constructive - the descriptions can be used as algorithms for evaluating the functions.

Following each function are examples of its use. All examples assume zero origin. Frequent lapses into English are included in an attempt to convey the spirit of the notation and give some insight into the why of functions as well as the what.

I. Mixed Functions

Any function which is not a scalar function is called a mixed function. Two important subsets of the mixed functions can be isolated. The structure functions [Morrow 17] generally manipulate entire arrays without regard to the value of the array. The select functions extract particular elements or subarrays of an array without regard to the type of the elements. The operands of these sets of functions can be classified as one of two types. An operand manipulated by a structure function or disected by a select function is called a subject operand. An operand which is used to further specify the action of a function is called a control operand. This classification of operands will make extension of the functions to new domains easy. Each of the mixed functions will be treated individually. The structure and select functions will be identified when they are introduced and are summarized in tables 1 and 2 on page 130.

Size (I) (structure function)

Syntax: $\underline{Z} \leftrightarrow \rho \underline{R}$

$\underline{Z}$ is a simple vector whose components represent the number of scalars along each coordinate of $\underline{R}$.

Conformability: $\underline{R}$ is the subject operand and is any array

The size function is the *APL* equivalent of the word dimension and $\rho \underline{R}$ is sometimes read dimension of $\underline{R}$. The compound function $\rho\rho \underline{R}$ is the *APL* equivalent of the word rank since the number of coordinates in an array is the number of elements in its dimension vector. The size function is useful in defining other functions. For example in the discussion of the monadic scalar functions the words "producing a result having the same dimension as the operand" could be replaced by $\rho \underline{Z} \leftrightarrow \rho \underline{R}$. This will be done from here on.

Reshape (I) (structure function)

Syntax: $Z \leftrightarrow L \rho R$

The result is an array of dimension L whose elements (if any) are taken from R in row major order, reusing elements of R if necessary.

Conformability: L is the control operand and is a non-negative integer scalar or a non-negative integer vector. R is the subject operand and is any array having at least one element unless a zero occurs in L in which case R may be empty.

Reshape is used to generate arrays of given dimension from scalars. The examples of scalar functions contain statements which use a meta-notation for arrays:

Example:

```
      0 > 3.14  .7
          4   ¯3.14
```

With reshape this could be written:

```
      0 > 2 2ρ 3.14  .7  4  ¯3.14
```

Ravel (I) (structure function)

Syntax: $Z \leftarrow\rightarrow ,R$

Z is the vector of the scalars in R taken in row major order.

Conformability: R is the subject operand and is any array.

Formal description: origin free

$Z \leftarrow\rightarrow (PROD\ \rho R)\ \rho\ R$

In the following descriptions of functions, ravel is used in two ways. First as a formal way to treat scalars as one-element vectors, and second to display how results are calculated when the structure of operands is not relevant to the calculation. An extension of ravel will be introduced later.

Examples:

```
      ,2 2ρ 'ABCD'
ABCD

      ρ,2 2ρ 'ABCD'
ρ1
4
```

This is the first example where the dimension display is required. It emphasizes the fact that the size function always results in a vector, never a scalar.

Vector Indexing (A) (select function)

Syntax: $Z \leftarrow L \; ⫤ \; R$

Z is an array of elements selected from positions L in R.

Conformability: R is the subject operand and is any vector.

L is a control operand and is any simple integer array.

Formal description: origin independent

$Z \leftarrow$

IF $0 = \rho\rho L$ *THEN*

IF $ORG \leq L < ORG+\rho R$

THEN the scalar which is the Lth component of R

ELSE IF $1=\rho\rho L$ *THEN* Z of dimension ρL such that

$I \; ⫤ \; Z \leftarrow (I \; ⫤ \; L) \; ⫤ \; R$

for each scalar integer I, $ORG \leq I \leq ORG+\rho R$

ELSE $(\rho L) \; \rho \; (,L) \; ⫤ \; R$

Vector indexing is fundamental to the definitions of the mixed functions. Most other select functions can be expressed in terms of indexing.

Examples:

```
      2 ⫤ 1 5 7 9
7

      (2 2⍴ 1 2 1 0) ⫤ 1 5 7 9
5 7
5 1
```

Vector Attach (I) (structure function)

Syntax: $\underline{Z} \leftrightarrow \underline{L},\underline{R}$

$\underline{Z}$ is a vector made by joining $\underline{L}$ to $\underline{R}$.

Conformability: $\underline{L}$ and $\underline{R}$ are subject operands and may be any vector or scalar.

Formal description: origin independent

$\underline{Z}$ of dimension $(\rho,\underline{L})+\rho,\underline{R}$ such that

$I \; \underline{Z} \leftrightarrow I \; ,\underline{L}$ for $\underline{QRG} \leq I < \underline{QRG}+\rho,\underline{L}$

$(I+\rho,\underline{L}) \; \underline{Z} \leftrightarrow I \; ,\underline{R}$ for $\underline{QRG} \leq I < \underline{QRG}+\rho,\underline{R}$

Vector attach (and its extension to array operands yet to be introduced) is the fundamental way of joining two arrays. In *APL\360* this function is called catenate but here that term shall be given a more restrictive meaning (see Catenate page 114)

Examples:

```
      1,2
1 2

      2 3,4
2 3 4

      1,2,3,4
1 2 3 4
```

Every simple array A can be represented by a compound function on scalars using only the primitive functions

vector attach and reshape. A vector can be represented as a compound function as follows:

1.) The empty vector is 0ρ0

2.) A one element vector is 1ρ$\underline{S}$ for $\underline{S}$ any scalar

3.) A vector with N elements is $S1,S2,\ldots,SN$ for any scalars Si

Given that a vector can be represented, any simple array A is represented by

(ρA)ρ,A

Where ρA is a vector and ,A is a vector.

Thus the statement is verified.

The representation of a numeric vector constant (i.e. 1 2 3 4) is really a primitive functional notation for the compound function 1,2,3,4.

Interval (I)

Syntax: *Z* ↔ ⍳*R*

Z is the length *R* vector of successive integers starting at the index origin.

Conformability: *R* is any scalar integer such that 0 ≤ *R*.

Formal description: origin independent

```
Z ↔
IF R = 0 THEN E
ELSE (⍳R-1),R+ORG-1
```

Examples:

```
      ⍳3
0 1 2

      ⍳0
E
```

Identities

(I1) $,\underline{L} \leftrightarrow \rho\underline{L}\rho\underline{R}$

(I2) $\underline{R} \leftrightarrow (\rho\underline{R})\rho\underline{R}$

(I3) $\rho\underline{L} \leftrightarrow \rho\ \underline{L}\ \exists\ \underline{R}$

(I4) $,\underline{S} \leftrightarrow \rho\iota\underline{S}$

(I5) $\underline{V} \leftrightarrow ,\underline{V}$

Vector Take (I) (select function)

Syntax: $Z \leftrightarrow L \uparrow R$

Z is a dimension ,L vector of the first or last L elements of R

Conformability: R is the subject operand and is any scalar or vector, L is a control operand and is any numeric scalar.

Formal description: origin free

$Z \leftrightarrow$

IF $L \geq 0$ *THEN*

IF $L \leq \rho ,R$ *THEN* $(\iota L) \boxed{9} ,R$

ELSE $R,(L-\rho ,R)\rho\theta$

ELSE IF $L < 0$ *THEN*

IF $L \geq -\rho ,R$ *THEN* $((\iota |L)+(\rho ,R)-L) \boxed{9} R$

ELSE $((|L+\rho ,R)\rho\theta),R$

where θ is the previously defined position scalar.

ELSE undefined

Examples:

```
 2 ↑ 1 2 3
1 2

 ¯3↑'IFATE'
ATE

 4 ↑ 3,'A'
3 A 0 0
```

Vector Drop (I) (select function)

Syntax: $Z \leftrightarrow L \downarrow R$

Z is R with the first or last L elements deleted.

Conformability: R is the subject operand and is any scalar or vector, L is a control operand and is any numeric scalar.

Formal description: origin free

```
Z ↔
IF (0 ≤ |L) AND ((|L)≤ ⍴,R)
   THEN (L-(×L)×⍴,R)↑R
ELSE E
```

Examples:

```
      2↓ 1 2 3
⍴1
3

      ¯3 ↓ 'IFATE'
IF

      4 ↓ 3,'A'
E
```

The definitions of take and drop illustrate the statement that functions which select elements from vectors can be defined in terms of vector indexing. The statement will remain true when indexing and the select functions are

defined for array operands. Of course none of the select functions are really needed since indexing could always be used. They are convenient in that they represent easily understood and often needed special cases of indexing and are often less clumsy to use.

For example given vector V, $1\uparrow V$ is not much different from $0 \text{ ⍋ } V$ except that the former is origin free and the latter is a scalar. $\bar{}1\uparrow V$ is nicer than $(\bar{}1+\rho V) \text{ ⍋ } V$.

Vector reduction (I)

Syntax: *Z* ↔ *D*/*R*

Z is a scalar derived from repeated applications of the scalar function *D* on scalars in *R*

Conformability: *R* is any scalar or vector, *D* is any dyadic scalar function.

Formal description: origin independent

```
Z ↔
IF E ↔ ⍴R THEN R
ELSE IF 0 = ⍴R THEN
  IF identity element I exists for D THEN I
  ELSE undefined
ELSE IF 1 = ⍴R THEN  E⍴R
ELSE (ORG ⍎ R) D D/1↓R
```

Examples:

```
      +/1 2 3
6

      -/1 2 3
2

      =/0 1,'AB'
1
```

+/ 1 2 3 can be considered a shorthand notation for (1+(2+3)).

Reduction could have been defined so that it would associate to the left and be shorthand for ((1+2)+3). This is the reduction originally defined by Iverson [Iverson 10] and the two definitions are obviously equivalent for any associative function. It is not the right to left rule that directs the choice of right association. Rather the reverse is true. Non-associative functions produce uninteresting numbers when evaluated from left to right. 1-2-...-*N* would be 1 minus the sum of the remaining numbers whereas when evaluated form right to left the same expression yeilds an alternating sum.

Reduction can be considered a device for defining dyadic scalar functions on other than two operands. Everyone knows that addition can be applied to more than two numbers. An algorithm is provided for adding sets of numbers. For all associative functions the meaning of reduction is well known. +/ is summation, ×/ is product, ⌈/ is biggest, etc.. When *D* is not associative reduction is still well defined and some still have well known meanings. -/ is the alternating sum, but ⍟/ is uncommon. ×/ is the *APL* equivalent for the meta-notation *PROD X* for vector *X*. The equivalence will remain when reduction is extended to arrays.

Vector Scan (I)

Syntax: $Z \leftrightarrow D\backslash R$

Z is a vector of partial reductions of elements in R.

Conformability: R is any vector, D is any dyadic scalar function.

Formal description: origin independent

Z of dimension ρR such that

$I \ni D\backslash R \leftrightarrow D/(I+1-ORG)\uparrow R$

for scalar integer I, $ORG \leq I < ORG+\rho R$

Examples:

```
      +\1 2 3
1 3 6

      -\1 2 3
1 ¯1 2

      =\0 1,'AB'
0 0 1 1
```

Vector back-scan (A)

Syntax: $\underline{Z} \leftrightarrow \underline{D}\backslant\underline{R}$

$\underline{Z}$ is a vector of partial reductions of elements in $\underline{R}$.

Conformability: $\underline{R}$ is any vector, $\underline{D}$ is any dyadic scalar function.

Formal description: origin independent

$\underline{Z}$ of dimension $\rho\underline{R}$ such that

$I \ni \underline{D}\backslant \underline{R} \leftrightarrow \underline{D}/(I-\underline{QRG})\downarrow\underline{R}$

for scalar I, $\underline{QRG} \leq I < \underline{QRG}+\rho\underline{R}$

Examples:

```
      +⌿1 2 3
6 5 3

      -⌿1 2 3
2 ¯1 3

      =⌿0 1,'AB'
1 0 0 B
```

Vector Base Value (Decode) (I)

Syntax: $Z \leftrightarrow L \perp R$

Z is a scalar which is the evaluation of R in the mixed radix L.

Conformability: L and R are any scalar or vector such that $\rho L \leftrightarrow \rho R$ or one of them is a scalar or a one element vector.

Formal description: origin free

$Z \leftrightarrow$

IF $,1 \leftrightarrow \rho L$ *THEN* $(E \rho L) \perp R$

ELSE IF $,1 \leftrightarrow \rho R$ *THEN* $L \perp E \rho R$

ELSE IF $\rho L \leftrightarrow \rho R$ *THEN* $+/R \times 1 \downarrow (\times \backslash L), 1$

ELSE undefined

Examples:

```
      10 ⊥ 1 2 3
123

      10 10 10 ⊥ 2
222

      4 2 3 ⊥ 2 1 0
15
```

Vector Represent (Encode) (I)

Syntax: $Z \leftarrow\rightarrow L \top R$

Z is the mixed base L representation of R.

Conformability: L is any simple numeric vector, R is any numeric scalar.

Formal description: origin free

```
Z ←→
IF  E ←→ ⍴L  THEN E
ELSE IF  ¯1↑L ←→ 0  THEN ((¯1↓L)⊤0),R
ELSE ((¯1↓L)⊤(R-(¯1↑L)|R)÷¯1↑L),(¯1↑L)|R
```

Examples:

```
      2 2 2 ⊤ 5
1 0 1

      3 4 2 ⊤ 5
 0  2  1

      ¯2 ¯2 ¯2 ⊤ 5
¯1 ¯1 ¯1
```

Vector Compress (I) (select function)

Syntax: $Z \leftrightarrow L/R$

Z is a a vector of scalars from R in positions corresponding to ones in L.

Conformability: L is a control operand and R is the subject operand. They may be any vector or scalar.

Formal description: origin independent

```
Z ↔
IF SCALAR L THEN
   IF L THEN ,R
   ELSE E
ELSE IF SCALAR R THEN L/(ρL)ρR
ELSE IF ρL ↔ ρR THEN
   IF 0 ↔ ρL THEN E
   ELSE ((ORG ⍳ L)/ORG ⍳ R),(1↓L)/1↓R
ELSE undefined
```

Note how the description is meaningless in case L is not composed of zeros and ones. In case L is scalar the result is all of R or none of it as L is a one or a zero.

Examples:

```
      1/'CAT'
CAT

      0/4 5 6
E
```

1 0 1/5 7 9

5 9

1 0 1/5

5 5

Vector Expand (I) (select function)

Syntax: *Z* ↔ *L**R*

Z is *R* expanded to positions corresponding to ones in *L*

Conformability: *L* is a control operand and *R* is the subject operand. They may be any vector or scalar.

Formal description: origin free

```
Z ↔
IF SCALAR L THEN
   IF ,1 ↔ ⍴,R THEN
      IF L THEN ,R
      ELSE undefined
   ELSE undefined
ELSE IF SCALAR R THEN L\(+/L)⍴R
ELSE IF ⍴R ↔ +/⍴L THEN
   IF 0 ↔ ⍴L THEN E
   ELSE IF 1↑L THEN (1↑R),(1↓L)\1↓R
   ELSE 0 ,(1↓L)\R
ELSE undefined
```

Identity:

(I6) *R* ↔ *L*/*L**R* for vector operands

Examples:

```
      1 0 1\5
5 0 5
      1 0 1 \ 5 7
5 0 7
```

Vector Reverse (I) (select function)

Syntax: $Z \leftarrow \phi R$

Z is the vector R with elements reversed.

Conformability: R is the subject operand and is any vector

Formal description: origin independent

$$Z \leftarrow ((\bar{}1+\underline{Q}R\underline{G}+\rho R)-\iota\rho R) \boxplus R$$

Example:

```
      ⌽⍳3
2  1  0
```

Vector Rotate (I) (select function)

Syntax: $Z \leftrightarrow L\phi R$

Z is the vector R with elements cyclicly rotated.

Conformability: R is the subject operand and is any vector.

L is the control operand and is any scalar integer.

Formal description: origin free

$Z \leftrightarrow ((\rho R)|L+\iota''\rho\rho R) \boxminus R$

Later when interval (ι) is extended to vector operands the sub-expression $\iota''\rho\rho R$ may be replaced by $\iota\rho R$.

Example:

```
      ¯1φ'ABC'
CAB
```

J. General Arrays

A simple array has been defined as a set of rank 0 arrays arranged along coordinates. A general array is the obvious extension of a simple array and is a set of rank *N* arrays arranged along coordinates. The definition of a general array includes that of a simple array. The personality of these new objects is determined by the functions which are defined upon them. The desire is to define the existing functions on this enlarged domain so that the useful identities are retained.

A graphic representation of arrays is now developed as an aid in discovering the properties of general arrays. The new representation is that of a singly rooted tree with labeled nodes. A singly rooted tree with labeled nodes (called simply a tree from here on) is a pair of sets $(N\ L)$ where N is a finite set of labeled nodes, and L is a finite set of ordered pairs of elements of N called lines or branches. The first element or each ordered pair is called the initial node, the second the final node. Trees have the following properties

1.) There is a distinguished node called the root which is the final node for no line.
2.) No node is the final node of more than one line.
3.) For every node Ni, either Ni is the root or

there exists a sequence of lines $L1, L2, L3, \ldots, LN$ such that the root is the initial node of $L0$ and Ni is the final node of Ln. Ni is said to be at the Nth-level of the tree.

4.) Every node of a tree defines a subtree of which it is the root.

The following terminology is used when talking about trees. A node which is the initial node for no line is called a leaf. A tree is simple if every node is either the root or a leaf. If a tree has a node at the Nth level but no node at the N+1st level, then it is called an N-level tree. A simple tree is either a 0-level or a 1-level tree. Figure 1 page 64 pictures a 2-level tree with 4 leaves.

Now a mapping is made from arrays to trees.

A scalar A is mapped to a tree $T(A)$ consisting only of its root. The root is labeled with the dimension and value of the scalar. (see Figure 2 page 64)

Thus a scalar is represented by a 0-level tree. Notice that the size function produces the dimension label of the root node.

Just as scalars are used to build arrays, scalar trees shall be used to build array trees. The extension from

scalars to simple arrays is made by going from 0-level trees to 1-level trees whose leaves are 0-level trees.

A simple array A is mapped to a simple tree $T(A)$ as follows:

1.) The root of the tree is labeled ρA.

2.) The root is the initial node for $\underline{PROD}\ \rho A$ lines.

3.) The final nodes of the $\underline{PROD}\ \rho A$ lines are leaves (making the tree simple) and are the scalars of A in ravel order.

(see Figure 3 page 65)

Notice that as before the size function ρ produces the dimension label of the root node. Since each leaf is a scalar tree, they are each labeled with a dimension so that identity (I3) on vector indexing is preserved. Using the second example of figure 3:

$'B' \leftrightarrow 1\ \text{⌷}\ A$ (definition of vector indexing)

$\rho 1\ \text{⌷}\ A \leftrightarrow \rho 1$ (by I3)

$\leftrightarrow \underline{E}$ (definition of a scalar)

which is the label on the node.

Each node also has a value. In the case of a leaf, the scalar value appears as the second label on the node. (This is consistent with the previous definition of a scalar tree). The value of the root node is the entire array which the node defines and as such is redundant and may be elided. The dimension part of the label is not strictly needed

because the tree for a rank N array could be drawn in $N+1$ space. This is not done because of the difficulty in visualizing the difference between the tree for ,3 in 2 space and the tree for 1 1⍴3 in 3 space. Thus a simple array is a 0-level or a 1-level tree whose leaves are scalar trees.

The obvious way to make the extension from simple arrays to general arrays is to examine more general trees.

Consider the tree required for a two element vector A whose first element is 'B' and whose second element is the three element vector ⍳3 (see Figure 4a page 66). It is not clear what label should be supplied for the unlabeled node. Property 4 of trees says that this node is the root of a subtree so the label must be 3. Yet if the node is viewed as the second element of a vector then to satisfy identity I3 the dimension label should be

⍴ 1 ⌷ A ↔ ⍴1 by I3

↔ E by def. of a scalar

Therefore to satisfy all the requirements the labels for nodes must be modified to contain two dimensions. First the dimension seen by functions (called the <u>apparent dimension</u>) and second the real dimension of the array (called the <u>hidden dimension</u>). Now a general array can be represented as in Figure 4b page 66.

The tree (and the array) defined by the node labeled $\underline{E}:3:$ has the property that it looks like a scalar yet has many values. Any tree whose apparent dimension is $\underline{E}$ is called a unit tree and the array it represents is called a unit array. Notice that the mapping from arrays to trees is not surjective (onto) but is injective (one-to-one). In particular no tree which represents an array can have a node whose apparent dimension is different from its hidden dimension unless it is a node defining a unit tree. It is possible that an attempt to make the mapping surjective (isomorphism) could lead to more general arrays. Further modifications of the labeling of these trees and/or permitting multiply rooted trees could also lead to extensions.

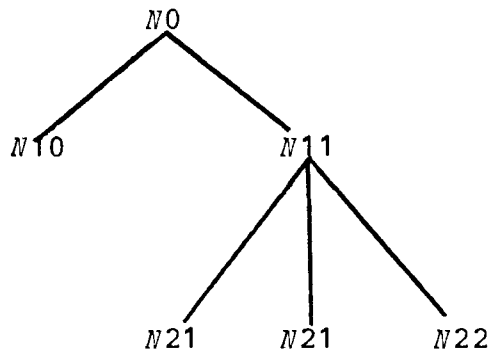


Figure 1

A singly rooted tree with labeled nodes

A	$T(A)$
3	$\underline{E}$: 3
'A'	$\underline{E}$: 'A'

Figure 2

Scalar Trees

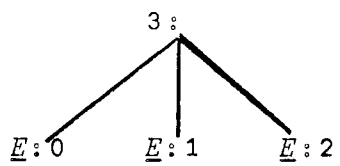
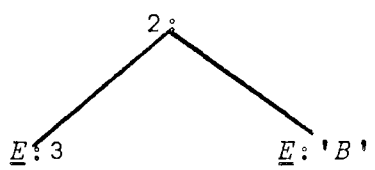
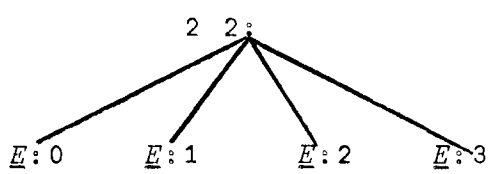
A	$T(A)$
⍳3	
3,'B'	
2 2⍴⍳4	

Figure 3

Simple trees

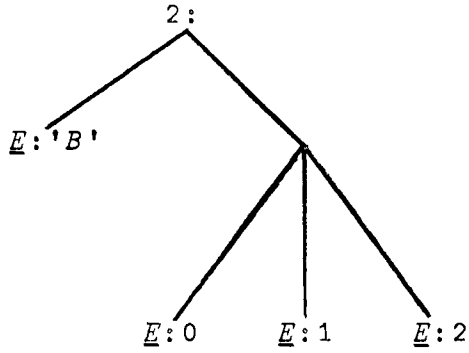


Figure 4a

Incomplete general tree

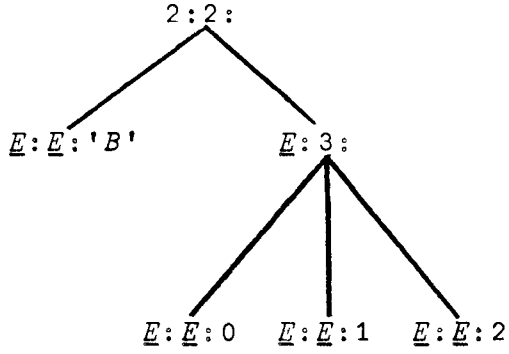


Figure 4b

General tree

K. Mixed Functions (continued)

Just as functions are needed to create simple arrays, they are needed to create general arrays. In this section the basic functions for creating and manipulating general arrays will be presented. Most of the mixed functions presented so far can be extended to the domain of general arrays by substituting the term unit array for the term scalar in their definitions. They will therefore not be presented again.

Conceal (A) (structure function)

Syntax: $\underline{Z} \leftrightarrow \subset\underline{R}$

$\underline{Z}$ is the unit array of the array $\underline{R}$.

Conformability: $\underline{R}$ is the subject operand and is any array.

Since $\subset\underline{R}$ is a unit array, the following identity is obvious

(I7) $\underline{E} \leftrightarrow \rho\subset\underline{R}$

The structure of $\underline{R}$ is not lost but merely not apparent (i.e. hidden from the view of some functions - dimension, reshape, ravel, etc.). Conceal is nilpotent on unit arrays and scalars because neither the dimension nor the values are changed by application of the function.

Identity:

(I8) $\underline{U} \leftrightarrow \subset\underline{U}$

In terms of the tree representation, conceal sets the apparent dimension of the root node to '$\underline{E}$'.

Reveal (A)

Syntax: $\underline{Z} \leftrightarrow \supset \underline{R}$

$\underline{Z}$ is the unit array $\underline{R}$ with its hidden dimensions revealed.

Conformability: $\underline{R}$ is the subject operand and is any unit array.

Identities:

(I9) $\underline{R} \leftrightarrow \supset \subset \underline{R}$ Reveal is the left inverse of conceal

(I10) $\underline{U} \leftrightarrow \subset \supset \underline{U}$

Thus $\underline{Z}$ is the array which is concealed in the unit array $\underline{R}$. In discussion of functions, reveal is used to exhibit the action of functions on general arrays, given that their action on simple arrays is known. Reveal is nilpotent on scalars.

Identity:

(I11) $\underline{S} \leftrightarrow \supset \underline{S}$

(proof)

$\underline{S} \leftrightarrow \supset \subset \underline{S}$ by I9

$\leftrightarrow \supset \underline{S}$ by I8

In terms of the tree representation, reveal changes the apparent dimension from $\underline{E}$ to the hidden dimension.

The *SCALAR* predicate may be defined by using reveal and the conditional as follows [Lathwell 12]

```
SCALAR R ←→
IF 0←→ρρR THEN
   IF 0←→ρρ⊃R THEN 1
   ELSE 0
ELSE 0
```

The expression $R \leftrightarrow \supset R$ which is true for scalars is not an appropriate definition because $\supset R$ may be undefined if R is not a unit array. An expression for the *SCALAR* predicate which does not involve the conditional is presented on page 90.

Unit Indexing (A) (select function)

Syntax: $\underline{Z} \leftrightarrow \underline{L}$ ⍧ $\underline{R}$

$\underline{Z}$ is an array of unit arrays selected from positions $\underline{L}$ in $\underline{R}$.

Conformability: $\underline{R}$ is the subject operand and is any array, $\underline{L}$ is a control operand and is any 0-level or 1-level array of integers.

Formal description: 0 origin dependent

$\underline{Z} \leftrightarrow$

$\underline{IF}$ 0 = ⍴⍴$\underline{L}$ $\underline{THEN}$ ((⍴$\underline{R}$)⍳⊃$\underline{L}$) ⍧ ,$\underline{R}$

$\underline{ELSE}$ $\underline{IF}$ 1 = ⍴⍴$\underline{L}$ $\underline{THEN}$ $\underline{Z}$ of dimension ⍴$\underline{L}$ such that

I ⍧ $\underline{Z} \leftrightarrow$ (I ⍧ $\underline{L}$) ⍧ $\underline{R}$

for I, $0 \leq I \leq$ ⍴$\underline{R}$

$\underline{ELSE}$ (⍴$\underline{L}$) ⍴ (,$\underline{L}$) ⍧ $\underline{R}$

Notice how the use of ⍳ in the definition imposes restrictions on $\underline{L}$. If $\underline{L}$ is not simple in line 2 then ⊃$\underline{L}$ is not a scalar and therefore must have dimension ⍴⍴$\underline{R}$. Thus a rank-N array is indexed by unit arrays which are the conceal of N element vectors. In particular a vector is indexed with unit arrays which are the conceal of one element vectors. In vector indexing scalars are used to index vectors. This is no problem because in case $\underline{L}$ is simple and $\underline{R}$ is a vector, the definition of unit indexing reduces to that of vector indexing. This is seen because the definitions are identical except when 0=⍴⍴$\underline{L}$. In that case $\underline{L}$ is scalar and:

```
   ((ρR)⊥⊃L) ⌹ ,R          def. of unit indexing
↔ ((ρR)⊥⊃L) ⌹ R           by I5
↔ ((ρR)⊥L) ⌹ R            by I11
↔ (+/L×1↓(×⍀ρR),1) ⌹ R    def. of ⊥
↔ (+/L×1↓(ρR),1) ⌹ R      def. of ×⍀
↔ (+/L×1) ⌹ R             def. of ↓
↔ L ⌹ R                   def. of +/
```

Therefore unit indexing is a proper extension of vector indexing.

Examples:

```
      ((⊂0 0),⊂1 2) ⌹ 3 3 ρ⍳9
0 5

      (⊂E) ⌹ 9
9
```

Unit indexing allows selection of elements from any array. In particular it allows indexing of unit arrays (and therefore scalars) yielding the following identity:

(I12) $U \leftrightarrow (\subset E) \mathbin{⌹} U$

(proof):

```
   (⊂E) ⌹ U
↔ ((ρ⊂E)⊥⊃⊂E) ⌹ ,U          by def. of ⌹
```

$\leftrightarrow$ $(\underline{E}\perp\supset\subset\underline{E})$ ⍎ $,\underline{U}$ by I7

$\leftrightarrow$ $(\underline{E}\perp\underline{E})$ ⍎ $,\underline{U}$ by I9

$\leftrightarrow$ $(+/\underline{E}\times 1\downarrow(\times\backslash\underline{E}),1)$ ⍎ $,\underline{U}$ by def. of $\perp$

$\leftrightarrow$ $(+/\underline{E}\times 1\downarrow\underline{E},1)$ ⍎ $,\underline{U}$ by def. of $\backslash$

$\leftrightarrow$ $(+/\underline{E}\times 1\downarrow 1)$ ⍎ $,\underline{U}$ by def. of ,

$\leftrightarrow$ $(+/\underline{E}\times\underline{E})$ ⍎ $,\underline{U}$ by def. of $\downarrow$

$\leftrightarrow$ $(+/\underline{E})$ ⍎ $,\underline{U}$ by the scalar extension

$\leftrightarrow$ 0 ⍎ $,\underline{U}$ by def. of $+/$

$\leftrightarrow$ $\underline{U}$ by def. of vector indexing

Identity:

(I13) J ⍎ K ⍎ $L \leftrightarrow (J$ ⍎ $K)$ ⍎ L

This is merely a substitution for $\underline{Z}$ in line 4 of the description and says that unit indexing is associative.

An index I to an array $\underline{R}$ is called a proper index (PI) of $\underline{R}$ if it selects a single unit array from $\underline{R}$. Formally I must be the conceal of a length $\rho\rho\underline{R}$ vector of integers such that

$\underline{QRG} \leq (J$ ⍎ $\supset I) \leq {}^{-}1+\underline{QRG}+J$ ⍎ $\rho\underline{R}$

for each scalar integer J

$\underline{QRG} \leq J \leq {}^{-}1+\underline{QRG}+J$ ⍎ $\rho\underline{R}$

In particular the only proper index of a unit array is $\subset\underline{E}$. The terminology PI will be used to eliminate the explicit declaration of the range of indexes.

L. Display of General Arrays

The display of a general array is not as straight forward as the display of a simple array because any index position may contain an arbitrarily complicated sub-array. The following algorithm is used for display of general arrays:

1.) If *A* is simple then display it as before.

2.) If *A* is not simple, then display in row major order as part of the dimension display, the index to each position in the array and then recursively display the reveal of the array in that position.

For example the array represented by the tree in figure 4b page 66 is displayed as follows:

```
0⍴⍴
B
1⍴⍴3
0  1  2
```

The array is a two element vector and is not simple. The first element is at index position zero and is a scalar so its dimension display is empty. The index positions and dimension are separated by the symbol ⍴. The second element

is at index position 1 and is a three element vector. This form is a little cumbersome and in an implementation it might be desireable to elide some of the detail. It is, however, an unambiguous and complete representation of a general array. The following properties may be read immediately from the dimension display:

1.) The dimension display of a simple array has no occurence of the symbol :

2.) The dimension display for a unit array begins with the symbol :

No primitive notation is provided for writing general arrays just as no way is provided for writing simple arrays or even scalar rational fractions.

M. Mixed functions (continued)

Entire (A) (select function)

Syntax: $Z \leftarrow \partial R$

Conformability: R is the subject operand and is any array.

Formal description: origin free

$Z \leftarrow R$

This is not a useful function at this time but later the fact that it is size preserving will be useful.

Same (A)

Syntax: $\underline{Z} \leftrightarrow \underline{L} \equiv \underline{R}$

$\underline{Z}$ is 1 if $\underline{L}$ and $\underline{R}$ are the same array and 0 otherwise.

Conformability: $\underline{L}$ and $\underline{R}$ may be any arrays

Formal description: origin free

```
Z ↔
IF (SCALAR L AND SCALAR R ) THEN L = R
ELSE IF (E ↔ ρL AND E ↔ ρR) THEN (⊃L) ≡ ⊃R
ELSE IF  ρL ↔ ρR
   THEN ∧/T  for T of dimension ×/ρL such that
   I ⌷ T ↔ (I ⌷ ,L) ≡ I ⌷ ,R
   for each PI I of ,L
ELSE 0
```

Examples:

```
      E ≡ 1 2 3
0

      E ≡ E
1
```

$\equiv$ is the *APL* equivalent of the meta-notation $\leftrightarrow$ and could be used in its place.

Membership (I)

Syntax: $\underline{Z} \leftrightarrow \underline{L} \in \underline{R}$

$\underline{Z}$ is one or zero as the corresponding element of $\underline{L}$ does or does not occur in $\underline{R}$

Conformability: $\underline{L}$ and $\underline{R}$ may be any arrays

Formal description: origin free

$\underline{Z}$ of dimension $\rho\underline{L}$ such that

$I \; \exists \; \underline{Z} \leftrightarrow \underline{IF} \; (I \; \exists \; \underline{L}) \equiv J \; \exists \; \underline{R}$

for some PI J of $\underline{R}$ $\underline{THEN}$ 1

$\underline{ELSE}$ 0

Examples:

```
      'CAT' ∈ 'CAB'
1 1 0

      1 3 5 ∈ ⊂1 3 5
0 0 0

      1 3 5 ∈ 3,⊂1 3 5
0 1 0
```

Complement-of (set Difference) (A)

Syntax: $Z \leftrightarrow L\sim R$

Z is the vector of elements from R which do not occur in L

Conformability: L and R may be any arrays.

Formal description: origin free

$Z \leftrightarrow (,\sim R \in L)/,R$

$L\sim R$ may be read as the complement of L in R. Notice that the structure of the operands is of no consequence.

Examples:

```
      (⍳3) ~ ⍳5
3 4

      'AB' ~ 1 2
1 2
```

The function may be used to delete the blanks in a character vector.

```
      ' '~'NOW IS THE TIME'
NOWISTHETIME
```

Index-of (ranking) (E)

Syntax: $Z \leftarrow L \iota R$

Z is the array of index positions for elements of R in L

Conformability: L and R may be any arrays

Formal description: origin free

Z of dimension ρR such that

$I \boxplus ,Z \leftarrow$

IF $E \leftarrow \rho L$ *THEN*

IF $L \leftarrow I \boxplus R$ *THEN* $\subset E$

ELSE θ

ELSE IF $I \boxplus ,R \leftarrow J \boxplus ,L$ for some scalar J

THEN $\subset(\rho L)\tau J$

for the smallest J such that $I \boxplus ,R \leftarrow J \boxplus ,L$

ELSE θ

Examples:

```
      1 2 3 ι 2 2ρ 3 1 0 3
0 0:ρ1
2

0 1:ρ1
0

1 0:ρ
θ
```

```
1 1:ρ1
2

      (2 2ρ'ABCD')ι'CA'
0:ρ2
1 0

1:ρ2
0 0

      E ← 5 6
0 0
```

This use of the position scalar 0 corresponds to the use of the null character ∘ in Iverson's discussion of ranking. [Iverson 10]

In certain special cases index-of may be considered an inverse to unit indexing.

Identity:

(I14) $R \leftrightarrow (R\iota R) \in R$

Transpose (I)

The definitions of dyadic and monadic transpose are so closely related that it is convenient to treat them together.

Syntax: $\underline{Z} \leftrightarrow \underline{L} \lozenge \underline{R}$ (or $\underline{Z} \leftrightarrow \lozenge \underline{R}$)

$\underline{Z}$ is $\underline{R}$ with permuted coordinates some of which may be aligned.

Conformability: $\underline{R}$ is any array, $\underline{L}$ in the dyadic case is any vector of elements taken from $\iota\rho\rho\underline{R}$ such that

$\rho\underline{L} \leftrightarrow \rho\rho\underline{R}$

$\wedge/\underline{L} \in \iota\rho\rho\underline{R}$

$\wedge/(\iota\lceil/\underline{L}) \in \underline{L}$ ($\underline{L}$ is dense)

[Abrams 1]

Dyadic Transpose

Formal description: origin independent

$\underline{Z}$ of rank $1\text{-}\underline{QRG}\text{-}\lceil/\underline{L}$ such that

$I \ni \rho\underline{Z} \leftrightarrow \lfloor/(\underline{L}=I)/\rho\underline{R}$

for each I, $\underline{QRG} \leq I \leq \lceil/\underline{L}$

and

$I \ni \underline{Z} \leftrightarrow (\subset\underline{L} \ni \supset I) \ni \underline{R}$

for each PI I of $\underline{Z}$

Monadic Transpose

Formal description: origin free

$Z \leftrightarrow (\phi \iota \rho \rho R) \lozenge R$

This description differs from that used in *APL\360* but is the transpose defined in the formal description of *APL* [Lathwell,Mezei 11]

Examples:

```
      1 0 ⍉ 2 3⍴'ABCDEF'     (or ⍉ 2 3⍴'ABCDEF')
AD
BE
CF

      0 0 ⍉ 2 3⍴'ABCDEF'
AE
```

In case L is a permutation of $\iota \rho \rho R$ or in every case of monadic transpose Z is said to be equivalent to R up to a transpose. When defining functions there are often many choices for the arrangements of the dimension vector. Each of the arrangements is equivalent up to a transpose to each of the others.

Identity:

(I15) $R \leftrightarrow (\iota \rho \rho R) \lozenge R$

Grade Up (I)

Syntax: $\underline{Z} \leftrightarrow \blacktriangle \underline{R}$

$\underline{Z}$ is the permutation of $\iota\rho\underline{R}$ which will order $\underline{R}$ from smallest to largest.

Conformability: $\underline{R}$ is any numeric vector. The position scalar θ is permitted in the domain of grade up and is considered smaller than any number.

Example:

```
      (▲5) ⍋ 5
5

       (▲ 5 θ 6 ¯2) ⍋ 5 θ 6 ¯2
θ ¯2 5 6
```

Grade up has the following useful property. If $\underline{R}$ is any permutation of ιN for some non-negative integer N then $\blacktriangle\underline{R}$ is its inverse permutation and $(\blacktriangle\underline{R}) \grave{\jmath} \underline{R} \leftrightarrow \iota N$ which is the identity permutation. Thus $\underline{R} \leftrightarrow \blacktriangle\blacktriangle\underline{R}$.

Grade Down (I)

Syntax: $Z \leftarrow \Psi R$

Z is the permutation of $\iota\rho R$ which will order R from largest to smallest.

Conformability: R is any numeric vector. The position scalar θ is permitted in the domain of grade down and is considered larger than any number.

In case R has no occurance of θ the following definition holds.

$Z \leftarrow \Delta\text{-}R$

Examples:

```
      (Ψ 5 θ 6 ¯2) ⌷ 5 θ 6 ¯2
θ 6 5 ¯2
```

Information about the Grade functions as implemented in *APL\360* may be found in the IBM Systems Journal [Woodrum 22].

N. Scalar Extension on General Arrays

In this section several methods for applying functions to general arrays are defined. The most fundamental of these is the scalar extension for scalar functions. A direct generalization of this is the scalar product operator which extends the same concept to non-scalar functions. Reduction has already been defined but is generalized. The outer product is defined as an alternate way to pair up the elements of the operands for application of the primitive definitions of functions. Inner product combines arrays using two primitive functions.

The Scalar Extension for Scalar Functions

The following definitions of scalar functions on general arrays assume only the primitive definitions of the functions. The extension of the functions to simple arrays is restated (now formally) providing a self-contained description.

1.) Scalar monadic functions: $Z \leftarrow M\ R$

```
Z ←
IF SCALAR R
   THEN  M R  (the primitive definition)
ELSE IF E ← ⍴R
```

THEN $\subset M \supset R$

ELSE *Z* of dimension ρR such that

I ⌷ *Z* ↔ *M I* ⌷ *R* for each *PI I*.

2.) Scalar dyadic functions: *Z* ↔ *L D R*

Z ↔

IF (*SCALAR L AND SCALAR R*) *THEN* *L D R*

(the primitive definition)

ELSE IF *E* ↔ ρL *THEN*

IF *E* ↔ ρR *THEN* $\subset(\supset L)$ *D* $\supset R$

ELSE $((\rho R)\rho L)$ *D R*

ELSE IF E ↔ ρR *THEN* *L D* $(\rho L)\rho R$

ELSE IF ρR ↔ ρL *THEN Z* of dimension ρL such that

I ⌷ *Z* ↔ (*I* ⌷ *L*) *D I* ⌷ *R*

for each *PI I* of *R*

ELSE undefined

Notice that a unit array is made dimensionally conformable to every array.

0. Operators

The scalar extension for scalar functions is certainly one of the important ways to apply functions to arrays. But it is only one of many possible extensions some of which are sufficiently useful to deserve primitive existence along with scalar extension. A function cannot be allowed to act

in more than one way on array operands so new functions are needed to represent the alternate array extensions. To invent new function symbols for them is no problem but is inefficient of symbols and confusing. Therefore objects called operators are introduced.

An operator is an object which takes functions as operands and produces a function as its result. Operators differ from functions as follows:

1.) They are of higher precedence than functions. The rightmost operator in an expression is evaluated before any function is evaluated.
2.) Both the left and/or right operands of an operator have limited scope. That is parentheses are not needed to limit the extent of the right operand.

It is interesting that an operator and its operands could be considered a multi-position symbol for a function (this has been done in implementations to date). This stand was taken implicitly in the earlier discussion of the reduction and scan functions. The symbols +/ may be taken to be a 2 position notation for a function which is related to addition (i.e. an alternate extension of addition on arrays). Or / may be considered a dextri-monadic operator which modifies its function operand + to be a new monadic function called plus reduction.

P. The Scalar Product Operator (A)

The scalar product operator is a monadic operator which has three definitions, one each when it is applied to monadic and dextri-monadic functions and one when it is applied to dyadic functions. It represents a method for applying functions to the subarrays concealed in general arrays. The symbol ⍮ is called demi-colon.

1.) Monadic functions: $Z \leftarrow\ ⍮\ M\ R$

$Z \leftarrow$

IF $E \leftarrow \rho R$ *THEN* $\subset M \supset R$

ELSE Z of dimension ρR such that

$I\ ⌷\ Z \leftarrow\ ⍮\ M\ I\ ⌷\ R$

A similar description applies to dextri-monadic functions.

2.) Dyadic functions: $Z \leftarrow L\ ⍮\ D\ R$

IF $E \leftarrow \rho L$ *THEN*

IF $E \leftarrow \rho R$ *THEN* $\subset (\supset L)\ D\ \supset R$

ELSE $((\rho R)\rho L)\ ⍮\ D\ R$

ELSE IF $E \leftarrow \rho R$ *THEN* $L\ ⍮\ D\ (\rho L)\rho R$

ELSE IF $\rho L \leftarrow \rho R$ *THEN* Z of dimension ρL such that

$I\ ⌷\ Z \leftarrow (I\ ⌷\ L)\ ⍮\ D\ I\ ⌷\ R$

ELSE undefined

Unlike scalar extension for scalar functions, recursion ends when a unit array is reached. In case $\underline{D}$ (or $\underline{M}$) is a scalar function then the operator is nilpotent. The operator is called the scalar product operator because the functions it produces act like scalar functions on the outermost level of their operands. A more general description of scalar extension will be given in a later section.

Examples:

```
      ⍀ ⍴ ⊂2 3⍴⍳6
:⍴2
2 3


      (⊂ 1 2) ⍀ , ⊂3 4
:⍴4
1 2 3 4


      (⊂ 4 5) ⍀ ≡ (⊂'AB'),⊂4 5
0  1
```

The scalar product operator allows definition of the *SCALAR* predicate:

SCALAR $\underline{R}$ ↔ (⊂⍳0) ≡ ⍀ ⍴$\underline{R}$

Q. Other Operators

Generalized Vector Reduction Operator (I)

Vector reduction is a dextri-monadic operator whose left operand is any dyadic function. Notice there is no restriction that this function be scalar or even primitive. It represents a way to apply a dyadic function *D* to a single array operand.

Syntax: *Z* ↔ *D*/*R*

Conformability: *R* is any vector or scalar, *D* is any dyadic function.

Formal description: origin independent

```
Z ↔
IF E ↔ ⍴R THEN R
ELSE IF 0=⍴R THEN
   IF identity element I exists for D THEN I
   ELSE undefined
ELSE IF 1=⍴R THEN  E⍴R
ELSE  (ORG ⍷ R) ⌿ D D/1↓R
```

Scan and backscan would be generalized by using this definition of reduction.

Example

```
      ⊃,/(⊂0 1),(⊂2 3),⊂4 5
0  1  2  3  4  5
```

Matrix Product Operator

The matrix product operator (∘.) is a dyadic operator. When both operands are dyadic functions, the resulting function is called inner product. When the right operand is a dyadic function and the left operand is the symbol ∘, the resulting function is called outer product. They each represent alternate ways to apply dyadic functions to array operands. The extension of inner product to arrays to be introduced later represents a generalization to the algebraic matrix product.

Outer Product (I)

Syntax: $Z \leftarrow L \circ . D\ R$

Z is the array produced by applying the dyadic function D to all pairs of elements, one from L, one from R

Conformability: L and R are any arrays, D is any dyadic function.

Formal description: origin free

Z of dimension $(\rho L),\rho R$ such that

$(I; ,J) \equiv Z \leftarrow (I \equiv L) ; D\ J \equiv R$

for each *PI* I of L and each *PI* J of R

Outer product may be used to generate operation tables for functions.

```
      (⍳3)∘.+⍳3
0  1  2
1  2  3
2  3  4

      (2,⊂1 3) ∘.+ 20,⊂30 40
0 0:⍴
2 2

0 1:⍴2
3 2 4 2

1 0:⍴2
2 1 2 3

1 1:⍴2
3 1 4 3
```

In case $\underline{L}$ and $\underline{R}$ are simple, the outer product may be expressed in either of the forms:

$\supset(\subset\underline{L})\ \underline{D}\ \underline{R}$

$\supset\underline{L}\ \underline{D}\ \subset\underline{R}$

Yet another form is discussed in Chapter 3 under scalar extension.

Vector Inner Product (I)

Syntax: $\underline{Z} \leftrightarrow \underline{L}\ \underline{D}.\underline{D}'\ \underline{R}$

Conformability: $\underline{L}$ and $\underline{R}$ are any vector or unit array. $\underline{D}$ and $\underline{D}'$ are any dyadic functions.

Formal description: origin free

$\underline{Z} \leftrightarrow \underline{D}/\ \underline{L}\ ⍪\ \underline{D}'\ \underline{R}$

$\underline{Z}$ is always a unit array because vector reduction always produces a unit array. Inner product will be extended to arrays along with other vector functions in section T of this chapter.

Operator examples

For all operators the functions may be any function (i.e. even non-primitive). In particular they may be functions resulting from an application of these operators. Thus these operators may be used to generate any number of new functions related to, but different from, the primitive function. The following examples take the same two operands (2,⊂1 3) and ((20,⊂30 40). The first case shows an ordinary attach. The next two show attach with the scalar product operator and with the outer product operator. The next four show the same operators on attach to two levels in all possible combinations.

```
      (2,⊂1 3),(20,⊂30 40)
0:⍴
2

1:⍴2
1 3

2:⍴
20

3:⍴2
30 40
```

```
      (2,⊂1 3) ⍪ ,(20,⊂30 40)
0:⍴2
2 20

1:⍴4
1 3 30 40

      (2,⊂1 3)∘.,(20,⊂30 40)
0 0:⍴2
2 20

0 1:⍴3
2 30 40

1 0:⍴3
1 3 20

1 1:⍴4
1 3 30 40
```

```
      (2,⊂1 3) ⍝ ⍝ ,(20,⊂30 40)
0:⍴2
2 20

1:0:⍴2
1 30

1:1:⍴2
3 40

      (2,⊂1 3)∘.∘.,(20,⊂30 40)
0 0:⍴2
2 20

0 1:0:⍴2
2 30

0 1:1:⍴2
2 40

1 0:0:⍴2
1 20

1 0:1:⍴2
3 20
```

```
1 1:0 0:ρ2
1 30

1 1:0 1:ρ2
1 40

1 1:1 0:ρ2
3 30

1 1:1 1:ρ2
3 40

      (2,⊂1 3)∘. ⍴ ,(20,⊂30 40)
0 0:ρ2
2 20

0 1:0:ρ2
2 30

0 1:1:ρ2
2 40

1 0:0:ρ2
1 20
```

```
1 0:1:⍴2
3 20

1 1:0:⍴2
1 30

1 1:1:⍴2
3 40

      (2,⊂1 3) ⍪ ∘.,(20,⊂30 40)
0:⍴2
2 20

1:0 0:⍴2
1 30

1:0 1:⍴2
1 40

1:1 0:⍴2
3 30

1:1 1:⍴2
3 40
```

R. Another Mixed Function

Index Generator (Interval) (Odometer) (E)

Syntax: $\underline{Z} \leftrightarrow \iota\underline{R}$

$\underline{Z}$ is the array of all valid indices to an array of dimension $\underline{R}$

Conformability: $\underline{R}$ is any vector of non-negative integers.

Formal description: 0 origin dependent

$\underline{Z} \leftrightarrow$

IF $0 \leftrightarrow \rho\underline{R}$ *THEN* $\subset\underline{E}$

ELSE $(\iota 0 \jmath \underline{R}) \circ. \, , \iota 1 \downarrow \underline{R}$

$\iota 0 \jmath \underline{R}$ is the previously defined interval function and $\iota 1 \downarrow \underline{R}$ is recursively the index generator. It is obvious from the use of the outer product that $\rho\underline{Z} \leftrightarrow \rho\underline{R}$.

Examples:

```
      ⍳3
0  1  2

      (⍳3) ⍷ ⍳5
0  1  2

      ⍳,3

0:⍴1
0
```

```
1:ρ1
1

2:ρ1
2

      (ι,3) ⍷ ι5
0  1  2

      ι2 3
0 0:ρ2
0 0

0 1:ρ2
0 1

0 2:ρ2
0 2

1 0:ρ2
1 0

1 1:ρ2
1 1

1 2:ρ2
1 2
```

Extended interval is called index generator for the obvious reason. The term odometer was used for a similar function defined by Abrams [Abrams 1] probably because the elements taken in row major order count in a base $\underline{R}$ number system.

An index I is a proper index (PI) of an array $\underline{R}$ iff $I \in \iota\rho\underline{R}$ and this may be considered an alternate definition for a proper index. Again note that for $\underline{R}$ a unit array

$$\iota\rho\underline{R} \leftrightarrow \iota\underline{E} \leftrightarrow \subset\underline{E}$$

Identities:

(I16) $\underline{R} \leftrightarrow (\iota\rho\underline{R})\ \exists\ \underline{R}$

(I17) $\underline{R} \leftrightarrow \rho\iota\underline{R}$ (easily proven by induction on $\rho\underline{R}$)

5. Indexed Functions

An array $\underline{R}$ of rank $N \geq 1$ may be thought of as any one of N sets of rank $N-1$ arrays joined along a new coordinate. For example a matrix of dimension 3 5 may be considered as 3 5-element vectors or 5 3-element vectors. In general an array $\underline{R}$ of rank $N \geq M$ may be thought of as any one of $M!N$ sets of rank $M-N$ arrays joined along M new coordinates.

Functions can be defined which act on array $\underline{R}$ as a single entity. But if one is thinking of $\underline{R}$ as a set of smaller rank arrays, it would be convenient to permit the functions to act on the smaller arrays as though they were the entities. Therefore many functions are equiped with a third operand called a function index which is used to specify the partitioning of the array. The function index (denoted $\underline{FI}$ in the meta-notation) is usually considered an index to the dimension vector of the right operand $\underline{R}$ of the function. As a vector index, its elements may be scalars or the conceal of one element vectors but for formal purposes the latter is assumed. An indexed function has its third operand displayed in brackets and appended to the right of the function symbol.

For functions which are defined on subarrays of an array it is convenient to allow elision of the index in case its value is $\iota\rho\rho\underline{R}$. Other indexed functions may have other defaults but in every case $\underline{FI}$ selects one or more

coordinates of a dimension. With each function will be a specification of the range of its function index.

used to index *M* and select the *I*th subarray.

Examples:

```
      ⊂[0] 2 3⍴⍳6
0 3

1 4

2 5

      ⊂[1] 2 3⍴⍳6
0 1 2

3 4 5
```

Identity:

(I18) $\subset\underline{R} \leftrightarrow \subset[\iota\rho\rho\underline{R}]\ \underline{R}$

(proof)

1.)dimension identity:

$\rho\subset[\iota\rho\rho\underline{R}]\ \underline{R}$

$\leftrightarrow ((\iota\rho\rho\underline{R})\sim\iota\rho\rho\underline{R})\ \exists\ \rho\underline{R}$ def. of ⊂[]

$\leftrightarrow \underline{E}$

Indexed Conceal (A) (structure function)

Syntax: $\underline{Z} \leftrightarrow \subset[\underline{FI}]\underline{R}$

$\underline{Z}$ is an array of unit arrays derived from subarrays of $\underline{R}$

Conformability: $\underline{R}$ is the subject operand and is any array. $\underline{FI}$ is any subset of $\iota\rho\rho\underline{R}$. The order of the elements in $\underline{FI}$ is relevant. If $\underline{FI}$ is elided it is taken to be $\iota\rho\rho\underline{R}$

Formal description: 0-origin dependent

$\underline{Z}$ of dimension $(\underline{FI}\sim\iota\rho\rho\underline{R})\ \lfloor\ \rho\underline{R}$ such that

$I\ \lfloor\ \underline{Z} \leftrightarrow \subset(I\ \natural\ ,\iota(-\rho\underline{FI})\uparrow\rho M)\ \lfloor\ M$

where

$M \leftrightarrow (\Phi((\iota\rho\rho\underline{R})\in\underline{FI})\backslash 1+\underline{FI})\Theta\underline{R}$

for each PI I of $\underline{Z}$

Although the description is involved, the function is simple. The indexed dimensions are hidden and the resulting subarrays are concealed. The dimension of each subarray is $\underline{FI}\ \lfloor\ \rho\underline{R}$. This means that the order of the elements in $\underline{FI}$ is important and that the subarrays concealed in $\underline{Z}$ are equivalent (up to a transpose) to the subarrays of $\underline{R}$. The transpose in the description moves the indexed dimensions in the order specified in $\underline{FI}$ to the right of the dimension vector defining the array M equivalent up to a transpose to $\underline{R}$. $\iota(-\rho\underline{FI})\uparrow\rho M$ is then the complete index set on those dimensions and defines a subarray. This index array is then united (via $\natural$,) in turn with each proper index I of $\underline{Z}$ and

Indexed reveal (A) (structure function)

Syntax: $\underline{Z} \leftrightarrow \supset[\underline{FI}]\ \underline{R}$

$\underline{Z}$ is the array $\underline{R}$ with its hidden dimensions revealed. Indexed reveal is the inverse of indexed conceal.

Conformability: $\underline{R}$ is the subject operand and is any array of unit arrays having the property

$\rho\supset I \fatsemi \underline{R} \leftrightarrow \rho\supset J \fatsemi \underline{R}$

for any PI's I and J of $\underline{R}$

$\underline{FI}$ is a vector of dimension $\rho\rho\supset I \fatsemi \underline{R}$ of vector indexes selected from the set $\iota(\rho\rho\underline{R})+\rho\rho\supset I \fatsemi \underline{R}$. If $\underline{FI}$ is elided it is taken to be $\iota\rho\rho\supset I \fatsemi \underline{R}$. The order of the elements in $\underline{FI}$ is relevant.

Formal description: 0 origin dependent

$\underline{Z}$ of rank $(\rho\rho\underline{R}) + \rho\rho\supset I \fatsemi \underline{R}$ such that

$\underline{FI} \fatsemi \rho\underline{Z} \leftrightarrow \rho\supset I \fatsemi \underline{R}$

$(\underline{FI}\sim\iota\rho\rho\underline{R}) \fatsemi \rho\underline{Z} \leftrightarrow \rho\underline{R}$

and

$\underline{Z} \leftrightarrow (\not\Downarrow\not\Downarrow((\iota\rho\rho M)\epsilon\underline{FI})\backslash 1+\underline{FI})\not\Diamond M$

for M such that

$(I\fatsemi ,J) \fatsemi M \leftrightarrow I \fatsemi \supset J \fatsemi \underline{R}$

Here M is generated with the hidden dimensions revealed on the right, then the inverse transpose of that used in indexed conceal is used to distribute them in the result dimension.

$\leftrightarrow \rho\subset\underline{R}$

2.)value identity:

$J\ \underline{\in}\ \subset[\iota\rho\rho\underline{R}]\ \underline{R}$

$\leftrightarrow J\ \underline{\in}\ \subset(J\underline{,}\ \iota(-\rho\iota\rho\rho\underline{R})\uparrow\rho M)\ \underline{\in}\ M$

for $M \leftrightarrow(\underline{\downarrow}((\iota\rho\rho\underline{R})\underline{\in}\iota\rho\rho\underline{R})\backslash 1+\iota\rho\rho\underline{R})\underline{\otimes}\underline{R}$

evaluating M

M

$\leftrightarrow(\underline{\downarrow}((\iota\rho\rho\underline{R})\underline{\in}\iota\rho\rho\underline{R})\backslash 1+\iota\rho\rho\underline{R})\underline{\otimes}\underline{R}$

$\leftrightarrow(\underline{\downarrow}1+\iota\rho\rho\underline{\rho})\underline{\otimes}\underline{R}$ by def. of \

$\leftrightarrow(\iota\rho\rho\underline{R})\underline{\otimes}\underline{R}$ by def. of $\underline{\downarrow}$

$\leftrightarrow \underline{R}$ by I15

using this result

$J\ \underline{\in}\ \subset[\iota\rho\rho\underline{R}]\ \underline{R}$

$\leftrightarrow J\ \underline{\in}\ \subset(J\underline{,}\ \iota(-\rho\iota\rho\rho\underline{R})\uparrow\rho\underline{R})\ \underline{\in}\ \underline{R}$ by def. of $\subset[]$

$\leftrightarrow(\subset\underline{E})\ \underline{\in}\ \subset((\subset\underline{E})\underline{,}\ \iota(-\rho\iota\rho\rho\underline{R})\uparrow\rho\underline{R})\ \underline{\in}\ \underline{R}$ by $J\in\iota\rho\rho\subset$exp.

$\leftrightarrow\subset(\iota(-\rho\iota\rho\rho\underline{R})\uparrow\rho\underline{R})\ \underline{\in}\ \underline{R}$ by I12 and def. of ,

$\leftrightarrow\subset(\iota(-\rho\rho\underline{R})\uparrow\rho\underline{R})\ \underline{\in}\ \underline{R}$ by I17

$\leftrightarrow\subset(\iota\rho\underline{R})\ \underline{\in}\ \underline{R}$ by def. of $\uparrow$

$\leftrightarrow\subset\underline{R}$ by I16

$\leftrightarrow(\subset\underline{E})\ \underline{\in}\ \subset\underline{R}$ by $J\in\iota\rho\rho\subset\underline{R}$

$\leftrightarrow J\ \underline{\in}\ \subset\underline{R}$

T. The General Array Extension Method

Indexed reveal and conceal along with the scalar product operator may be used to define a function on array operands when given its definition on vector operands. An array operand is concealed along an appropriate coordinate making it an array of one smaller rank whose elements are vectors (though not necessarily simple vectors) then the scalar product operator is applied to the functional so that its vector definition conforms with the hidden vectors. A more general form of this same scheme conceals more than one dimension and allows easy definition for the action of a function index on a function already defined on array operands. This is the case with indexed unit indexing.

This basic pattern will be used without further comment in many of the descriptions which follow. Understanding of the functions on vector operands is assumed so little discussion will accompany the function descriptions.

Examples:

```
      ⊃[0] (⊂0 3), (⊂1 4),⊂2 5

0 1 2

3 4 5


      ⊃[1] (⊂ 0 1 2), ⊂ 3 4 5

0 1 2

3 4 5
```

Identities:

(I19) $\underline{R} \leftrightarrow \supset[\underline{FI}]\subset[\underline{FI}]\ \underline{R}$

(I20) $\underline{R} \leftrightarrow \subset[\underline{FI}]\ \supset[\underline{FI}]\underline{R}$

```
      (0 ∘.⍴ ⍳ , 0 1 ∘.⍴ ⍳ , 0 3) ⌹ 3 3 3⍴⍳27
0  2
3  5
```

Note that the second outer product could have been written simply ∘., . If one defined a cartesian product function (;) having the definition

$L;R \leftrightarrow L \circ.\rho \iota , R$

then the above expression could be written

```
(0 ; 0 1 ; 0 3 ) ⌹ 3 3 3⍴⍳27
```

This form has a pleasing similarity to the bracket indexing of *APL\360* differing only in the possible need for extra parentheses.

Indexed Unit Indexing (A) (select function)

Syntax: $\underline{Z} \leftrightarrow \underline{L}$ ⌸ [$\underline{FI}$] $\underline{R}$

$\underline{Z}$ is a set of rank $(\rho\rho\underline{R})-\rho\underline{FI}$ subarrays selected from $\underline{R}$

Conformability: $\underline{R}$ is the subject operand and is any array, $\underline{L}$ is the control operand and is any 0-level or 1-level array of integers. $\underline{FI}$ is any subset of $\iota\rho\rho\underline{R}$ such that $\rho\supset I$ ⌸ $\underline{L} \leftrightarrow \rho\underline{FI}$ for any *PI* *I* of $\underline{L}$. If $\underline{FI}$ is elided it is taken to be $\iota\rho\rho\underline{R}$.

Formal description: origin free

$\underline{Z} \leftrightarrow \supset[\underline{FI}]\ (\subset\underline{L})$ ⍠ ⌸ $\subset[\underline{FI}]\ \underline{R}$

Identities:

(I21) $\underline{L}$ ⌸ $\underline{R} \leftrightarrow \underline{L}$ ⌸ [$\iota\rho\rho\underline{R}$] $\underline{R}$

(I22) $\underline{R} \leftrightarrow \underline{L}$ ⌸ [$\underline{E}$] $\underline{R}$

Examples:

```
      1 3 ⌸ [1] 3 4ρ'ABCDEFGHIJKL'
BD
FH
JL
```

This form of unit indexing eases the selection of cross sections of an array. More general cross sections may be taken by calculating an index as a cartesian product of the desired coordinates. Thus selecting plane 0, rows 0 and 1, columns 0 and 3 of `3 3 3ρι27` may be written

Attach (I)

Syntax: $\underline{Z} \leftrightarrow \underline{L},[\underline{FI}]\ \underline{R}$

$\underline{Z}$ is the array formed by joining $\underline{L}$ to $\underline{R}$.

The function index is either a scalar or the conceal of a one element vector and may be thought of as indicating the position in the result dimension along which the operands are joined. If $\underline{FI}$ is elided it is taken to be

$\lceil/\iota(\rho\rho\underline{L})\lceil\rho\rho\underline{R}$

There are three distinguishable cases of attach and for purposes of clarity each will be treated separately.

Catenate

Two arrays of equal rank are attached yielding an array of the same rank.

Conformability: $(\underline{FI}\neq\iota\rho\rho\underline{L})/\rho\underline{L} \leftrightarrow (\underline{FI}\neq\iota\rho\rho\underline{L})/\rho\underline{R}$

$\underline{FI}\in\iota\rho\rho\underline{L}$

Formal description: origin free

$\underline{Z} \leftrightarrow \supset[\underline{FI}]\ (\subset[\underline{FI}]\ \underline{L})\ ⍪\ ,\ \subset[\underline{FI}]\ \underline{R}$

Examples:

```
      1 2,[0] 3 4
1 2 3 4
      (2 3⍴⍳6),2 2⍴⍳4
0 1 2 0 1
3 4 5 2 3
```

Indexed Ravel (E) (structure function)

Syntax: $\underline{Z} \leftrightarrow ,[\underline{FI}]\ \underline{R}$

Conformability: $\underline{R}$ is the subject operand and is any array, $\underline{FI}$ is any permutation of $\iota\rho\rho\underline{R}$. If $\underline{FI}$ is elided it is taken to be $\iota\rho\rho\underline{R}$.

Formal description: origin free

$\underline{Z} \leftrightarrow ,FI\circledcirc\underline{R}$

The description is so nearly primitive itself that it is questionable if the function is worthy of primitive existence. However its inclusion removes some of the special status attributed to row major order.

Example:

```
      ,[1 0] 2 3ρ'ABCDEF'
ADBECF
```

```
1  2  3
```

```
      6,[0] 2 3⍴⍳6
6  6  6
0  1  2
3  4  5
```

Laminate

Arrays of identical shape are attached yielding an array of one higher rank.

Conformability: $\rho L \leftrightarrow \rho R$ or one array must be a unit array.

FI must be non-integer in the range $(QRG-1) < FI < QRG+\rho\rho R$

Formal description: origin free

$Z \leftrightarrow \supset[\lceil FI] \; L \; , \; R$

Examples:

```
      2,3
2  3
```

```
      'ABC',[¯.5] 'DEF'
ABC
DEF
```

```
      'ABC',[.5] 'DEF'
```

In the description of catenate each operand is concealed along the indexed dimension. These arrays are then conformable for the scalar product operator which is used to catenate the hidden vectors.

Adjoin

Arrays which differ in rank by one are attached yielding an array having the larger rank.

Conformability: $1 \leftrightarrow |(\rho\rho L) - \rho\rho R$ or one array must be a unit array.

$FI \in \iota(\rho\rho L)\lceil\rho\rho R$

Formal description: 0-origin dependent

$Z \leftrightarrow$

$IF\ (\rho\rho L) < \rho\rho R\ THEN\ \supset[FI]\ L\ ;\ ,\subset[FI]\ R$

$ELSE\ \supset[FI]\ (\subset[FI]\ L)\ ;\ ,\ R$

In adjoin the indexed dimension of the array of larger rank is concealed making the arrays conformable for the scalar product operator. An alternate definition would be to reshape the array of smaller rank to insert a one in the indexed dimension making the resulting arrays conformable for catenate.

Examples:

```
      1,2 3
```

Indexed Take (E) (select function)

Syntax: *Z* ↔ *L* ↑[*FI*] *R*

Conformability: *R* is the subject operand and is any array, *FI* is any subset of ⍳⍴⍴*R*. If *FI* is elided it is taken to be ⍳⍴⍴*R*. *L* is the control operand and is a simple vector such that ⍴*L* ↔ ⍴*FI*.

Formal description: origin independent

Z ↔

IF E ↔ ⍴*FI THEN R*

ELSE (1↓*L*)↑[1↓*FI*] ⊃[1↑*FI*] (*ORG* ⌷ *L*) ⍪ ↑ ⊂[1↑*FI*] *R*

When *FI* is elided, this is the same as the take function defined in *APL\360*.

Example:

```
      2↑[0] 3 3⍴⍳9
0  1  2
3  4  5

      ¯2 2 ↑[1 0] 3 3⍴⍳9
1  2
4  5
```

AD

BE

CF

Here unit arrays are joined by applications of the scalar product operator yielding hidden two element vectors. The fractional function index then indicates where in the result dimension the hidden dimension of 2 should be inserted.

The attach function defined by this triple of descriptions is essentially unchanged from the attach function defined in *APL*\360.

Base Value (decode) (I)

Syntax: $\underline{Z} \leftrightarrow \underline{L} \perp \underline{R}$

Conformability: $\underline{L}$ and $\underline{R}$ are any arrays such that

$$^{-}1 \uparrow \rho \underline{L} \leftrightarrow 1 \uparrow \rho \underline{R}$$

Formal description. origin independent

$$\underline{Z} \leftrightarrow (\subset[^{-}1 \uparrow \iota \rho \rho \underline{L}]\ \underline{L}) \circledast \perp \subset[\underline{Q}\underline{R}\underline{G}]\ \underline{R}$$

This function is identical with the Base value defined in *APL\360*.

Example:

```
      10 10 10 ⊥ 3 2 ⍴⍳6
24 135
```

Indexed Drop (E) (select function)

Syntax: *Z* ← *L*↓[*FI*] *R*

Conformability: same as indexed take.

Formal description. origin independent

Z ←

IF E ← ⍴*FI THEN R*

ELSE (1↓*L*)↓[1↓*FI*] ⊃[1↓*FI*] (*ORG* ⍳ *L*) ⌿ ↓⊂[1↑*FI*] *R*

When *FI* is elided this is the same as the drop function defined in *APL*\360.

Example:

```
      2 ↓[0] 3 3⍴⍳9
⍴1 3
6 7 8

      ¯2 2 ↓[1 0] 3 3⍴⍳9
⍴1 1
6
```

Indexed Compress (I) (select function)

Syntax: $\underline{Z} \leftrightarrow \underline{L}/[\underline{FI}]\ \underline{R}$

Conformability: $\underline{R}$ is the subject operand and is any array.

$\underline{FI} \in \iota\rho\rho\underline{R}$, if $\underline{FI}$ is elided it is taken to be $\lceil/\iota\rho\rho\underline{R}$.

$\underline{L}$ is a numeric vector such that $\rho\underline{L} \leftrightarrow \underline{FI} \not\equiv \rho\underline{R}$

Formal description: origin free

$\underline{Z} \leftrightarrow \supset[\underline{FI}]\ (\subset\underline{L})/\subset[\underline{FI}]\ \underline{R}$

This function is identical to the compress defined in *APL\360*.

Example:

```
      1 0 1/[0] 3 3ρι9
0  1  2
6  7  8
```

Represent (Encode) (I)

Syntax: $\underline{Z} \leftrightarrow \underline{L}\ \top \underline{R}$

Conformability: $\underline{L}$ and $\underline{R}$ are any numeric arrays.

Formal description: origin free

$$\underline{Z} \leftrightarrow \supset[^{-}1\uparrow\iota\rho\rho\underline{L}]\ \subset[^{-}1\uparrow\iota\rho\rho\underline{L}]\ \S\ \top\ \underline{R}$$

This function is identical with the represent defined in *APL\360*.

Example:

```
      10 10 10 ⊤ 24 135
0  1
2  3
4  5
```

Indexed Reverse (I) (select function)

Syntax: $Z \leftarrow\rightarrow \phi[FI]\ R$

Conformability: R is the subject operand and is any array.

FI is any subset of $\iota\rho\rho R$, if FI is elided it is taken to be $\iota\rho\rho R$.

Formal description: origin free

$Z \leftarrow\rightarrow$

$IF\ E \leftarrow\rightarrow \rho FI\ THEN\ R$

$ELSE\ \phi[1\downarrow FI]\ \supset[1\uparrow FI]\ ,\ \phi\ \subset[1\uparrow FI]\ R$

The first ϕ in this description is a recursive reference to indexed reversal, the second is vector reversal. This function differs from the reverse defined in *APL\360* in that any or all coordinates may be reversed, and when FI is elided, all coordinates are reversed.

Example:

```
      ϕ[1] 3 3ρι9
2 1 0
5 4 3
8 7 6
```

Indexed Expand (I) (select function)

Syntax: $\underline{Z} \leftrightarrow \underline{L} \backslash[\underline{FI}] \underline{R}$

Conformability: $\underline{R}$ is the subject operand and is any array.

$\underline{FI} \in \iota\rho\rho\underline{R}$, if $\underline{FI}$ is elided it is taken to be $\lceil/\iota\rho\rho\underline{R}$.

$\underline{L}$ is a numeric vector such that $+/\underline{L} \leftrightarrow \underline{FI} \ni \rho\underline{R}$

Formal description: origin free

$\underline{Z} \leftrightarrow \supset[\underline{FI}] (\subset\underline{L})\neq \backslash \subset[\underline{FI}] \underline{R}$

This function is identical to the expand defined in *APL\360*.

Example:

```
      1 0 1\[0] 2 3ρ0 1 2 6 7 8
0  1  2
0  0  0
6  7  8
```

It is interesting to note that if $\underline{FI}$ is permitted to be a subset of $\iota\rho\rho\underline{R}$, that the definition still holds. If this extension were adopted, then the following example would be valid:

```
      (3 3ρ0 1 1 1)\[0 1] 3 2ρι6
0  0  1
2  0  3
4  5  0
```

Indexed Reduction

Syntax: *Z* ↔ *D*/[*FI*] *R*

Conformability: *R* is any array, *FI* is any subset of ⍳⍴⍴*R*, if *FI* is elided then it is taken to be ⍳⍴⍴*R*, *D* is any dyadic function

Formal description: origin free

```
Z ↔
IF E ↔ ⍴FI THEN R
ELSE D/[¯1↓FI-FI>¯1↑FI] ⍎ D/⊂[¯1↑FI] R
```

The first *D*/ in this description is a recursive reference to indexed reduction, the second is vector reduction. The expression in the function index of *D*/ assures that throughout the recursion the index will refer to the original coordinates of the array *R*. This function differs from the reduction defined in *APL\360* in two ways. First any or all coordinates may be reduced. Second when no function index is supplied, reduction always produces a unit array.

Example:

```
      -/[1 2] 2 3 4⍴⍳24
0 0
```

Indexed Rotate (I) (select function)

Syntax: $\underline{Z} \leftrightarrow \underline{L}\phi[\underline{FI}]\ \underline{R}$

Conformability: $\underline{R}$ is the subject operand and is any array. $\underline{L}$ is the control operand and is an integer scalar or a simple array of integers of dimension $(\underline{FI}\neq\iota\rho\rho\underline{R})/\rho\underline{R}$. $\underline{FI}\in\iota\rho\rho\underline{R}$. If $\underline{FI}$ is elided it is taken to be $\lceil/\iota\rho\rho\underline{R}$.

Formal description: origin free

$\underline{Z} \leftrightarrow \supset[\underline{FI}]\ \underline{L}\ \S\ \phi\ \subset[\underline{FI}]\underline{R}$

This function is identical to the rotate defined in *APL\360*. It is particularly pleasing that the general array extension method applies so easily in the description of this function. $\underline{R}$ is concealed along the indexed dimension leaving an array conformable to a scalar $\underline{L}$ or an array $\underline{L}$ of like dimension.

Example:

```
      ¯1 0 1 ϕ 3 3ρι3
2 0 1
0 1 2
1 2 0
```

Inner Product

Syntax: $\underline{Z} \leftarrow \underline{L}\ \underline{D}\circ\underline{D}'\ \underline{R}$

Conformability: $\underline{L}$ and $\underline{R}$ are any arrays such that

$^{-}1\uparrow\rho\underline{L} \leftrightarrow 1\uparrow\rho\underline{R}$, $\underline{D}$ and $\underline{D}'$ are any dyadic functions.

Formal description: origin independent

$\underline{Z} \leftarrow$ ⍎ $\underline{D}/$ $(\subset[^{-}1\uparrow\iota\rho\rho\underline{L}]\ \underline{L})$ ⍎ $\underline{D}'$ $\subset[\underline{Q}\underline{R}\underline{G}]\ \underline{R}$

This function is the same as the inner product function defined in *APL\360*.

Indexed Scan

Syntax: $\underline{Z} \leftrightarrow \underline{D}\backslash[\underline{FI}]\ \underline{R}$

Conformability: same as indexed reduction

Formal description: origin independent

$\underline{Z} \leftrightarrow$

$\underline{IF}\ \underline{E} \leftrightarrow \rho\underline{FI}\ \underline{THEN}\ \underline{R}$

$\underline{ELSE}\ \underline{D}\backslash[\bar{}1\downarrow\underline{FI}] \supset[\bar{}1\uparrow\underline{FI}]\ ;\ \underline{D}\backslash\subset[\bar{}1\uparrow\underline{FI}]\ \underline{R}$

Indexed back-scan has a similar description by using backscan in place of scan in the above description.

name	syntax	*L*	default *FI*	*R*
reshape	*Z* ↔ *L*⍴*R*	control	---	subject
attach	*Z* ↔ *L*,[*FI*]*R*	subject	⌈/⍳(⍴⍴*L*)⌈⍴⍴*R*	subject
size	*Z* ↔ ⍴*R*	---	---	subject
ravel	*Z* ↔ ,[*FI*]*R*	---	⍳⍴⍴*R*	subject
conceal	*Z* ↔ ⊂[*FI*]*R*	---	⍳⍴⍴*R*	subject
reveal	*Z* ↔ ⊃[*FI*]*R*	---	⍳⍴⍴⊃*I* ⌷ *R*	subject

Table 1 The Structure Functions

name	syntax	*L*	default *FI*	*R*
indexing	*Z* ↔ *L*⌷ [*FI*]*R*	control	⍳⍴⍴*R*	subject
entire	*Z* ↔ ⌷ *R*	---	---	subject
take	*Z* ↔ *L*↑[*FI*]*R*	control	⍳⍴⍴*R*	subject
drop	*Z* ↔ *L*↓[*FI*]*R*	control	⍳⍴⍴*R*	subject
compress	*Z* ↔ *L*/[*FI*]*R*	control	⌈/⍳⍴⍴*R*	subject
expand	*Z* ↔ *L*\[*FI*]*R*	control	⌈/⍳⍴⍴*R*	subject
reverse	*Z* ↔ ⌽[*FI*]*R*	---	⍳⍴⍴*R*	subject
rotate	*Z* ↔ *L*⌽[*FI*]*R*	control	⌈/⍳⍴⍴*R*	subject
transpose	*Z* ↔ *L*⍉*R*	control	---	subject
transpose	*Z* ↔ ⍉*R*	---	---	subject

Table 2 The Select Functions

U. Remarks

A multitude of primitive functions have been presented in this chapter. The majority of these functions already exist in *APL\360* but nearly all have been redefined to some extent (the scalar functions being the notable exception). One could continue to define primitive functions for more and more specialized purposes but given a sufficiently powerful and general set of primitives these special functions may be defined as sets of compound expressions. To achieve this, notational existence must be attributed to some set of objects corresponding to function symbols and to *L*, *R*, *Z*, and *FI* of the meta-notation. These objects are called names. They and functions defined on them shall be the subject of Chapter 2.

The following page contains a summary of the structure and select functions to provide an easy reference for the definition of these functions on names.

any examples in this paper to avoid confusion with the meta-notation. An identifier is an atomic object and its multiposition display is not significant. An identifer cannot be confused with a character constant because there are no enclosing quotes. (This is a stronger reason for using quotes for character constants than the one given earlier.)

Examples:

X

RATE

L49

A name is any identifier or any function symbol and is therefore not an array. A name may be associated with a constant array called its description and the resulting pair is called a variable function. Variable functions which are named by function symbols are called primitive functions and have already been discussed. Variable functions which are named by identifiers are separated into three classes based on the type of the constant array. 1.) if the array does not contain a program scalar then the variable function is called a variable. 2.) If the array contains only program scalars, then the variable function is called a defined function. 3.) Any other variable function is called a mixed variable and is treated syntacticly as a variable. The term variable is used because at different times a name may be associated with different constant descriptions. This is not

Chapter 2

Defined Functions and the Name Domain

A. Introduction

All the examples of the preceding chapter involved primitive functions with literal operands. All the descriptions involved a meta-notation for operands with the understanding that any literal in the domain of a given function could be substituted for the meta-notation. If all possible functions were primitive, then no further machinery would be required. But as was stated before not all functions are created equal and only the useful and general ones are endowed with primitive existence. It becomes useful then to permit one to derive a new function from the old ones in terms of general operands as is done in the meta-notation and then apply the function as though it were primitive. To this end the concept of a variable which will play the role of the meta-notation for operands and functions is introduced.

B. The Name Domain

An identifier is a sequence of alphabetic (A-Z, and $\underline{A}$-$\underline{Z}$) and numeric (0-9) characters whose left-most symbol is an alphabetic. The underscored alphabetics are not used in

C. Functions in the Name Domain

Specification (I)

Syntax: $Z \leftrightarrow L \leftarrow R$

The description associated with the name L (if any) is replaced by R.

Conformability: L is any identifier, the position scalar 0, or the special symbol $\square$, R is any array.

Formal description: origin free

$Z \leftrightarrow R$

Specification is indeed a unique function. Its evaluation is trivial (much like entire) but it has the important side effect of either changing a name into a variable function, changing the description of a variable function (in which case it is often called a respecification), or in case $0 \leftrightarrow L$ having no effect at all. The clear distinction between the domains permits the inclusion of the p-scalar in the name domain without ambiguity. When $L \leftrightarrow \square$ a display of R is generated.

Examples:

```
      Z←⍳3
0 1 2

      B←'AB'
AB
```

the same as a variable in the mathematical sense. In the equation

$$((X*2)-Y*2)=(X+Y)\times(X-Y) \quad ,$$

if the variables ever denote values, they denote "any value". In *APL* a variable has one particular value (its description). The concept of a compound expression is extended to include variable functions and a variable function used as an operand to some other function always requires evaluation. The value of a variable function is the array produced when its description is evaluated. The evaluation of a variable is trivial and the value is not distinguishable from the description. The evaluation of a defined function implies evaluation of the program it represents and in general may depend upon other arrays (its operands).

An operand of a function is said to be in the value domain when the arrays of the operand are the objects of interest. This is the case with the operands of all functions introduced so far. An operand of a function is said to be in the name domain when the names occuring in the operand are themselves the objects of interest. It does not in general matter if the names are variables (i.e. are associated with descriptions) or not.

The obvious link between the value domain and the name domain is a function which has one operand in each domain and which establishes the association between a name and a description.

Chapter 3 Section B). The second restriction is that no defined function equivalent to specification is permitted.

Having established a link between the value domain and the name domain it is natural to explore the name domain further.

A name scalar is a dimensionless array in the name domain containing a single name. Specification is extended to allow $\underline{L}$ to be a name scalar and its action is imposed on the name composing the scalar. As with general arrays in the value domain the differing personalities between a name and a name scalar are determined by the functions which are defined on them. In the following pages the structure and the select functions shall be defined in the name domain. It is this latter group which recognizes a difference between a name and a name scalar.

In the value domain, after a scalar was introduced the first order of business was to define structured arrays of scalars. The same is done in the name domain. Name scalars arranged in order along coordinates form a name array which has name dimension. A unit name includes several names but has empty dimension. A general name array has unit names arranged in order along coordinates. As before, functions are needed to construct the name arrays out of name scalars. And again the problem of symbol pollution arises. New, different, exciting (and probably hard to remember) function symbols could be invented for functions in the name domain but should be avoided if possible. Fortunately functions

In these examples as in all previous examples the result produced by the function evaluation is displayed. It turns out to be convienent in an implementation to inhibit the display of a result in the case it is a result of the specification function. This convention will be followed from here on and is the reason for introducing the special □ symbol.

The fact that specification has $\underline{E}$ as a result means that multiple specifications are possible in a compound expression. Any such expression can be evaluated by the algorithm on page 30 without ambiguity.

```
      A←⍳B←0
      B
0
      A
E
```

Thus specification is used to supply a description for a name. A more general specification will be introduced shortly which permits specification of selected elements of a variable. The left operand of specification is considered to be in the name domain only in context of the specification arrow ←. This prompts treatment of specification as a special object rather than a function. Treating it as a function is only possible with two restrictions. First no monadic or niladic ← shall be permitted. If they were defined then a no-result expression left of ← would be interpreted in the wrong domain (see

These functions may be used in the obvious way to produce any general name array.

The important property of the structure functions in this domain is that the relevant structure of the subject operands is the dimension of the names. The names need not be variables; and if they are, the structure of their associated descriptions is utterly ignored. For this reason the structure functions do not differentiate a name from a name scalar and names are treated as name scalars.

defined in the value domain may be extended to the name domain without ambiguity (again because of the clear distinction between the domains). This is in effect trading off symbol creation for increased sensitivity to context.

Structure Functions in the Name Domain

Generally the extension of functions to the name domain involves restatement of the description used in the value domain with the term name array substituted for the term array. Therefore the descriptions will not be repeated in detail.

The following table summarizes the structure functions which are extended by allowing the subject operand(s) to be in the name domain. Notice that a control operand and a function index are always in the value domain.

name	syntax	*Z* domain	*L* domain	*FI* domain	*R* domain
reshape	*Z* ↔ *L*ρ*R*	name	value	----	name
attach	*Z* ↔ *L*,[*FI*]*R*	name	name	value	name
size	*Z* ↔ ρ*R*	value	----	----	name
ravel	*Z* ↔ ,[*FI*]*R*	name	----	value	name
conceal	*Z* ↔ ⊂[*FI*]*R*	name	----	value	name
reveal	*Z* ↔ ⊃[*FI*]*R*	name	----	value	name

TABLE 3. Structure Functions in the Name Domain

While interpreting the above description keep in mind that $\underline{L}$ is in the name domain and that any functions applied to $\underline{L}$ (i.e. ⊃ and ρ) must be deciphered in the name domain. This is the reason for the curious notation $\underline{L}$i. It is desired to select a name from $\underline{L}$ yet no select functions have yet been extended to the enlarged domain. General specification will play an important role in defined functions and in arrays of functions. It could be argued that the case ($\underline{E}$ ↔ ρ$\underline{L}$) *AND* ($\underline{E}$ ←/→ ρ$\underline{R}$) should be undefined but the definition given is reasonable and useful (see the second example).

Examples:

```
      (A,B)←0
      A
0
      B
0

      (⊂A,B)←⍳3
      A
0  1  2
      B
0  1  2
```

General Specification (E)

Syntax: $\underline{Z} \leftrightarrow \underline{L} \leftarrow \underline{R}$

The descriptions associated with the names $\underline{L}$ (if any) are replaced by the arrays in $\underline{R}$.

Conformability: $\underline{L}$ is any array of distinct names, $\underline{R}$ is any unit array or any array such that $\rho\underline{R} \leftrightarrow \rho\underline{L}$.

Formal description: origin free

$\underline{Z} \leftrightarrow \underline{R}$

Note that $\underline{L}$ is an indirect reference to the values associated with the names in $\underline{L}$. As with the specification defined earlier the result is not as important as the side effect of associating arrays (i.e. descriptions) with names. The following is the assignment rule which specifies how arrays in $\underline{R}$ are associated with names in $\underline{L}$.

IF SCALAR $\underline{L}$ *THEN* $\underline{L} \leftarrow \underline{R}$

(specification as previously defined)

ELSE IF $\underline{E} \leftrightarrow \rho\underline{L}$ *THEN*

IF $\underline{E} \leftrightarrow \rho\underline{R}$ *THEN* $\subset(\supset\underline{L}) \leftarrow \supset\underline{R}$

ELSE $(\supset\underline{L})i \leftarrow \underline{R}$

ELSE IF $\underline{E} \leftrightarrow \rho\underline{R}$ *THEN* $\underline{L} \leftarrow (\rho\underline{L})\rho\underline{R}$

ELSE IF $\rho\underline{L} \leftrightarrow \rho\underline{R}$ *THEN* $\underline{Z}$ of dimension $\rho\underline{R}$ such that

$i \not\exists \underline{Z} \leftrightarrow \subset(\supset\underline{L}i) \leftarrow \supset i \not\exists \underline{R}$

where $\underline{L}i$ is the unit name which occurs in index position i of $\underline{L}$.

ELSE undefined

Select Functions in the Name Domain

The select functions are extended to the name domain by allowing the subject operand to be in the name domain. Unlike the structure functions, the select functions are defined only for names having descriptions (i.e. variable functions) or for name arrays. In case the subject operand of a select function is a name (say N) associated with an array (say A), then it is treated as a simple name array defined as follows. The size of the name array is ρA. The names in the name array are the names of unit arrays in corresponding positions of A.

For example suppose $N \leftarrow 2\ 3\rho\iota 6$ is evaluated. Then the name array used when N is the subject operand of a select function could be pictured as follows:

```
N00   N01   N02
N10   N11   N12
```

Names as above are never really created but it is a convenient memory device for understanding how the functions evaluate. Notice that each name is associated with a position in N. Now the select functions may be defined assuming that the subject operand is a name array and then the definitions are the same as in the value domain with the term name array substituted for the term array. Therefore the descriptions shall not be repeated. A table of select

```
      (A,B)←⊂⍳3
      A
:⍴3
0  1  2
      B
:⍴3
0  1  2

      (2 2⍴ A,B,C,D)← 2 2⍴⍳4
      A
0
      B
1
      C
2
      D
4
```

The p-scalar may be used to force conformability.

```
      (E, ⍬ ,F)←⍳3
      E,F
0  2
```

The following expression can be used to directly interchange two unit arrays.

```
      (A,B)←B,A
```

```
Name Entire
      (⍞ A)←0
      A
0 0 0

Name Take
      (4↑A)←'ABCD'
      A
ABC
```

Note the use of the p-scalar in the name domain to form A0 A1 A2 θ

```
Name Drop
      (2↓A)←7
      A
0  1  7

Name Compress
      (1 0 1/A)←5 6
      A
5  1  6

Name Expand
      (1 1 0 1\A)←'ABCD'
      A
ABD
```

Note the use of the p-scalar

functions would be pointless because the right operand is always in the name domain and the function index and left operand (if any) are always in the value domain. Therefore each select function shall be discussed by example.

Assume the following specifications have been evaluated just prior to each example:

```
      A←⍳3
      B←2 2⍴⍳4
      C←2 2⍴⍳4
```

Name Unit Indexing

```
      (2 0 ⌷ A)← 5 6
      A
6 1 5
```

This example will be discussed in detail because all other select functions are defined in terms of unit indexing. First, *A* is a name and not a name array. Therefore it is treated as the name array which could be pictured

*A*0 *A*1 *A*2

The 2 0 name index of that is *A*2 *A*0. Then applying the general specification function is like applying the two expressions:

```
      A2←5
      A0←6
```

and the resulting value of *A* is as described.

```
      ((0 ⍪ [0]B),0 0 ⍴C)←0
      B
0 0
2 3
      C
0 1
2 0
```

```
Name Reverse
      (⌽A)←'ABC'
      A
CBA

Name Rotate
      (1⌽A)←'ABC'
CAB

Name Transpose
      (0 0 ⍉B)← 8 9
      B
8  1
2  9
```

Since the result of a select function is a name array, a compound expression on select functions is well defined.

```
      A←⍳10
      (3↑5↓A)← 91 92 93
      A
0  1  2  3  4  91  92  93  8  9
```

Combinations of select and structure functions are valid if the operands to each function are conformable and if the resulting name array is an array of distinct names.

may not in general be clear which expressions comprise the definition. This problem may partly be solved by supplying a notation which emphasizes the relationship of expressions. Therefore an expression separator symbol ◇ is introduced. An occurrence of this symbol on a line is to mean that the expressions so separated are related for purposes of definition but are separated for evaluation. The occurrence of a ◇ which does not separate two expressions is ignored. The previous example may now be written as follows:

```
      V← 12 6 7 3 15 10 18 5
      MEAN←(+/V)÷⍴V ◇ DSQ←(V-MEAN)*2 ◇ VR←+/DSQ÷⍴V
      VR
23.75
```

Now the definition is distinguished from the setting of its operand and the displaying of its result. Observe that the expressions of the definition have been evaluated in sequence proceeding from left to right. The decision to evaluate in this order is somewhat arbitrary. It emphasizes the fact that ◇ is punctuation and not a function. Choosing a right to left evaluation could tend to imply some special significance which does not exist.

Of the many remaining objections to this method of definition, three are outstanding. First a definition may be comprised of a large number of expressions and an attempt to make them colinear is neither practical nor desireable. Second, the variables which are operands and the variables

D. The Defining of Functions

Frequently the description of a function in *APL* notation requires a set of expressions evaluated in sequence. This is certainly true when defining specialized functions from the primitive ones. For example consider defining a function to calculate the variance *VR* of a vector of numbers *V*. It could be defined as the following set of expressions which may be divided into three groups. The first expression defines the (implied) operand of the function; the next three calculate the result; the last expression displays the result.

```
      V← 12 6 7 3 15 10 18 5
      MEAN←(+/V)÷⍴V
      DSQ←(V-MEAN)*2
      VR←+/DSQ÷⍴V
      VR
23.75
```

Of course the above definition could be expressed in one line but in general that is not always possible and even when it is the one-line syndrome tends to cloud a definition. A set of expressions such as the above is a perfectly valid method for calculating a variance. There are, however, some formal and pragmatic objections to be aired. First, there is no clear distinction between the definition of the function and its evaluation and in fact it

E. Defined Functions

A defined function is a variable function whose description is a program scalar defined by one or more compound expressions. Evaluation of a variable of type program then implies evaluation of these expressions and is the non-trivial case of variable evaluation mentioned earlier. Such a function has two properties of interest: its description, and the evaluation of its description for given operands.

The intent is to have defined functions indistinguishable from primitive functions as far as their behaviour is concerned so that the language becomes functionally extensible. This means that any mention of the function name (in the value domain) with appropriate operands must imply evaluation of the function. Yet for a defined function to be really valuable there must be a way to retreive the unevaluated description for inspection and manipulation. This implies making arrays of type program more tangible (i.e. in that they are in the range and domain of some functions).

One problem in including a new type to the notation is that the domain of each primitive function must be examined to see if it should be extended to include the new type. This matter can be disposed of quickly in this case. The structure and select functions are extended such that their subject operands may contain arrays of type program. No

which are intermediate results are not distinguished. Informal operands as these are called implied operands. Applying the function to many sets of operands implies repeating the definition that many times because the definition and its evaluation are inseparable. Lastly, there is no way to indicate a recursive evaluation.

Clearly what is needed is a way to associate a name with the expressions such that a mention of the name implies evaluation of the expressions. Such a variable function is called a defined function and is discussed next.

Define (A)

Syntax: $Z \leftrightarrow \nabla R$

Z is the program expression for the expressions represented in the character array R

Conformability: R is any character or program array.

Formal description: origin free

```
Z ↔
IF R of type program THEN R
ELSE IF 1≥ρρR THEN the scalar program expression for R
ELSE ∇ ,R,'□'
```

The program expressions are not evaluated and therefore need not be meaningful or syntacticly correct. There is in general no means provided to both define and evaluate a program expression in the same compound expression. However the description of a variable may be specified by one expression, then evaluated by another:

```
      V←∇'A←B'
      A←B←3 □ V
6
      B←4 □ V
7
```

Thus the define function together with specification may be used to assign to a variable a program description consisting of an arbitrary number of expressions which are called lines of the function. The lines are associated with

other functions are extended. Thus function perturbation caused by the new type is minimized. A further simplification is that no special notation is introduced for program constants. This means that every array having elements of type program arise from the evaluation of some expression. Therefore there is need of a function which will take an operand of some other type and evaluate to a program.

Definition-of operator (A)

Syntax: $\underline{Z} \leftrightarrow \Delta\underline{R}$

$\underline{Z}$ is the unevaluated program description of $\underline{R}$

Conformability: $\underline{R}$ is any single function (including the primitive functions)

The definition-of operator is only valuable if some of the elements of $\underline{R}$ are of type program (since otherwise the description and the value are identical). The operator is in no sense an inverse to define because it is an operator , and it evaluates to a program and not to characters. It is similar to the special form QUOTE of LISP [McCarthy,et.al. 16]

Examples on primitive functions

```
      MINUS←Δ-
      MINUS 5
¯5
      8 MINUS 5
3
```

This shows in a more dramatic way that the description of a (primitive or defined) function is sensitive to it's context.

successive integers called line numbers. The first expression is assigned line number 1. There seems to be no reason to make the numbering origin sensitive. Line numbers are significant in conjunction with the branch function yet to be introduced.

If the description of a function is to be computationally useful there must be a means to extract it from the function. It is clear that no function can perform this task because its operand would be the function whose description was desired. But the mere mention of the operand implies its evaluation prior to the evaluation of the proposed function. Therefore an operator is required.

While an unheaded function appears to have many shortcomings, it is a fundamental object in the notation. Every expression written in the notation may be considered a literal unheaded defined function having one line. Thus the expression

```
2+3
```

is a literal unheaded defined function having a result. A set of expressions written colinearly and separated by the expression separator symbol, ◇ ,may be considered a literal unheaded defined function having several lines.

Other than this, the principle value of an unheaded defined function is that it is always in the domain of the evaluate function yet to be introduced.

An Example Defined Function

Now it is possible to define a variable function which calculates a variance.

```
VAR←∇'MEAN←(+/V)÷⍴V ⍝ DSQ←(V-MEAN)*2 ⍝ VR←+/DSQ÷⍴V'
```

This statement defines the function but does not cause evaluation of the expressions. The same function could have been defined by using the character array *G* having the following description:

```
      G
MEAN←(+/V)÷⍴V
DSQ←(V-MEAN)*2
VR←+/DSQ÷⍴V
      VAR←∇G
```

Using this defined function, a variance may be calculated on many sets of data without repeating the definition. By convention the function value produced is the value (if any) of the last expression evaluated. Thus any function defined as above is niladic and returns a result if the last expression evaluated returned a result. This type is called an unheaded defined function to distinguish it from another type to be presented shortly.

```
      V←12 6 7 3 15 10 18 5
      VAR
23.75
```

G. Headed Defined Functions

Defined functions as presented in the last few sections provide a means for associating a name with an ordered set of expressions. The function thus produced is niladic because its operands may only be implied. The function may or may not produce a result; and if it does, the result must be the last value calculated in the function. A name which is specified within the function retains its assigned value after function evaluation is complete.

For example consider the function *VAR* previously defined. Its operand *V* is implied and must be assigned a value prior to evaluating *VAR*. The result *VR* must be (and is) calculated producing the function result. It is assigned to a name inside the function only to suppress a display of the result. The function result is a proper operand for other functions so a standard deviation could be calculated by the expressions:

```
      V←12 6 7 3 15 10 18 5
      (VAR)*.5
4.873397172
```

For the function *VAR*, the extra names used to hold intermediate results may be useful

```
      MEAN
9.5
```

F. Display of Defined Functions

The display of a program array is a display of each expression in the function. The form of the display is unspecified except that it must be an unambiguous representation of the expressions in the function. In an implementation it is convenient to display a function in a one expression per line format. It is also convenient to display the line numbers. This form shall be used in this paper. It may be useful to attempt a display which reflects the original character form of the function description if it is known. The definition-of operator produces the unevaluated program description of a function and is therefore the ordinary way to request a function display. The example of the previous section would be displayed as follows:

```
      ΔVAR
[1]   MEAN←(+/V)÷⍴V
[2]   DSQ←(V-MEAN)*2
[3]   VR←+/DSQ÷⍴V
```

A function is said to be locked if display of its description is inhibited. All primitive functions are locked. In an implementation it may be desirable to provide a mechanism (outside the language) which will lock a defined function.

2.) The names of the formal results and the manner in which their values form the actual result. A formal result is a name which potentially becomes associated with an array during function evaluation. When function evaluation is complete this array becomes (at least part of) the actual result.

3.) The valid context of the function. The arrangement of the formal parameter(s) in the header determines whether the function is dyadic, monadic, etc..

4.) Names which are bounded on this function and on referenced functions. These are called local names. A name is local to a function if its value (if any) exists only while the function is being evaluated. A name which exists outside the function is called a global name. A local name may be thought of as qualified by the function name in that it is distinguished from a global name having the same identifier. An action on the local name does not affect the global name and the global is said to be shadowed by the local. A local name in a function is global to any functions

In the general case the intermediate results may be of only passing interest within the function yet their value will remain after evaluation is completed.

These properties may be considered features or handicaps depending upon the desired use of the function. They are all features if the function is really a shorthand for entering expressions one by one. If however the criterion is to mimic the primitives then this type of defined function fails. Therefore a second type of defined function is introduced called a headed defined function.

A headed defined function is a defined function whose first expression is not subjected to ordinary evaluation but rather is a prototype expression which exhibits the essential features of the function. This first expression is called the function header and contains the following information.

1.) The names of the formal parameters and the manner in which actual parameters are associated with them. A formal parameter is the name of a general operand to the function. Its use is similar to the meta-notation for operands. An actual parameter is an array whose value becomes associated with the formal parameter during a particular application of the function.

A function header has the following syntax:

*E*0 ← *E*1 _ [*FI*] *E*2;*N*0;*N*1

where

*E*0 ↔ The expression of the formal result. If the function is to have no result *E*0 and ← are elided.

*E*1 ↔ The expression of the formal left operand. If the function is monadic or niladic *E*1 is elided.

_ ↔ The placeholder for the function name. This may not be elided. The occurance of this symbol distinguishes headed functions from un-headed functions.

FI ↔ The expression of the formal function index. If the function is not indexable, then [*FI*] is elided.

*E*2 ↔ The expression of the formal right operand. If the function is niladic or dextri-monadic *E*2 is elided.

*N*0 ↔ The list of local names separated by blanks. An integer occuring in this list of names is

referenced by this function.

5.) Names which are bounded on this function only. These are called strictly local names. A strictly local name has all the properties of a local name except that it is not global to referenced functions. That is a mention of the name in a referenced function refers to the global definition.

this function. The symbol _ may tend to become lost among the multitude of names and expressions in the header but it can be useful to envision filling in the blanks at every occurance of _ with the name of the variable associated with the function description.

*E*1 and *E*2 are of the form $\underline{L}$ or $\underline{L}\leftarrow\underline{R}$ where $\underline{R}$ (if included) is any valid expression in the value domain and $\underline{L}$ is any array of distinct names created by use of structure functions only. The names occuring in $\underline{L}$ are considered local unless they occur in the list *N*1 in which case they are strictly local. Upon evaluation of the function one of the following actions occurs for each permitted operand:

1.) If an operand is supplied (say *B*) then the specification $\underline{L}\leftarrow B$ is evaluated.

2.) If no operand is supplied but $\underline{R}$ is included then the specification $\underline{L}\leftarrow\underline{R}$ is evaluated. (i.e. $\underline{R}$ is a default operand)

3.) If no operand is supplied and $\underline{R}$ is not included then no specification is performed and the names in $\underline{L}$ remain undefined. Note that ← is still written.

taken to be a local origin (i.e. the index origin for this function and for referenced functions).

*N*1 ↔ The list of strictly local names separated by blanks. An integer occuring in this list of names is taken to be a strictly local origin (i.e. the index origin for this function only).

Clearly no name may be declared both local and strictly local, nor can more than one index origin be specified. Any variable appearing in a defined function which is neither local nor strictly local is a global variable to the function although it may be local to a function which referenced this function.

The operands *E*1, *FI*, and *E*2 are called the formal parameters of the function. Their arrangement about the _ determine the valid syntax for the function. When the function is to be evaluated, the name of the function is presented in context with operands called actual parameters. Generally the actual parameters are in the value domain and the formal parameters are in the value domain. Parameter substitution is then very much like specification except that the names in the formal parameters are local.

An occurance of the symbol _ in any expression other than the function header implies a recursive reference to

expression in the value domain. This expression is evaluated after all the program expressions in the description have been evaluated and the resulting value becomes the value of the function. Names occuring in *E*0 are considered local unless they occur in the list *N*1. The names have no

descriptions at the start of function evaluation unless they duplicate names appearing in *E1*, *FI*, and *E2*. The facility to default operands which are not supplied allows definiton of related functions of differing syntax in a single description.

Example headed defined functions

The previously defined function *VAR* which computes a variance is repeated below defined as a headed defined function. Recall that the function header is a prototype statement and not an expression and therefore does not have a line number. It is convenient however to call it line 0 and this shall be done. Note that all variables are local and the operand is actual so the function acts like a primitive monadic function.

```
      ∆VAR
[0]   VR← _ V;MEAN DSQ
[1]   MEAN←(+/V)÷⍴V
[2]   DSQ←(V-MEAN)*2
[3]   VR←+/DSQ÷⍴V

      VAR 12 6 7 3 15 10 18 5
23.75
```

The following example shows a function *MINUS* which acts like the primitive - and one which acts like reduction where the default function index is ⌈/⍳⍴⍴*R* instead of ⍳⍴⍴*R*.

```
      MINUS← ∇'(A-B)←(A←0) _ B'
      5 MINUS 3
2
      MINUS 6
¯6
```

This example shows how the formal result of a function is evaluated in the value domain. Recall that the header is not itself evaluated and that therefore the ←'s are not specifications.

```
      REDUCE ← ∇'Z←F _ [I←⌈/⍳⍴⍴A] A ⌷ Z←F/[I] A'
      Δ+ REDUCE 2 3⍴⍳6
3  12
```

Here the left operand of reduce is the unevaluated description of the function +. This description becomes associated with the name *F* during evaluation and *F/[I]A* is then plus reduction.

H. Branching in Defined Functions

The expressions of a defined function are evaluated in index order. This is a somewhat arbitrary (though convenient) choice. It is often desirable to specify some other ordering perhaps even one which causes repeated evaluation of some expressions. To this end a function is defined which is meaningful only when evaluated in the context of a defined function and which specifies a departure from the standard sequence of evaluation.

Branch (I)

Syntax: →*R*

This function has no result but specifies the line number of the expression in the defined function *D* to be evaluated next.

Conformability: *R* is any array.

In the following (somewhat less than formal) description, *N* is the largest line number in the defined function *D*.

IF 0 ∈ ρ*R* *THEN* evaluation of this expression continues

ELSE IF (*ORG*) ⌷ ,*R*) ∈ 1-*ORG*-ι*N* then evaluation of this expression is abandoned and evaluation of line (*ORG* ⌷ ,*R*) commences

ELSE function evaluation terminates with the evaluation of the function result (if any)

A common way to terminate function evaluation is →0.

A branch most commonly occurs at the left end of an expression in which case a branch to an empty array is like branching to the next expression in sequence. It is often bothersome to keep track of line numbers in a function especially when its description is being changed. Therefore another type of local object is defined called a label. A

label is a local constant which is named by an identifier. Any program expression in a defined function *D* (including the function header) may be of the form "identifier:expression" in which case the identifier is taken to be a local constant whose scalar integer value is the line number of the expression. As usual a label is local unless it appears in the list *N*1. Branching and labels are also permitted in unheaded defined functions and the labels are local. Notice that a label (and its :) is attached to an expression but not part of the expression. Later when rules for synthesizing expressions are presented, labels will not be considered.

I. Manipulating Function Descriptions

The definition-of operator when applied to a function produces a scalar of type program. This is not an appropriate result if the purpose is to modify the description rather than to display or rename it. Therefore a function is presented which produces the character representation of a scalar.

Character Form (A)

Syntax: $\underline{Z} \leftrightarrow \tau\underline{R}$

$\underline{Z}$ is a character array which represents the scalar $\underline{R}$.

If $\underline{R}$ is numeric, then $\underline{Z}$ is a vector; if $\underline{R}$ is a program, then $\underline{Z}$ is a matrix with one row per line. This function along with definition-of is an inverse to define up to equivalent expressions. Let $F \leftarrow \nabla G$ Then $\tau\Delta F$ is a character matrix which has the same expressions as G expressed in a one line per row format with each line in some cannonical form (i.e. some blanks deleted, etc.). The evaluation of the *APL* notation is not sensitive to the exact cannonical form and it is therefore not specified other than to say that it ought to match the display format.

Identity: for function F

$F \leftrightarrow \nabla\tau\Delta F$

Inclusion of this function is consistent with limiting the number of functions defined on the program domain. It permits modification of function descriptions using more general text handling functions. In an implementation it would be reasonable to disallow application of this function to locked functions. Its application to numbers gives a convenient way to do numeric to character conversions.

J. Defined Functions and Evaluate

There is generally no way provided to both define and apply a function at the same time. This may be attributed to the difficulty of syntactically determining the operands of the function. For more discussion on this topic see Chapter 3 Section B. In the special case where the function is niladic and therefore has no operands, immediate evaluation is possible. Therefore the following function is defined.

Evaluate (Execute) (Unquote) (?)

Syntax: $Z \leftrightarrow \perp R$

Z is the result (if any) of evaluating the program expression R.

Conformability: R is any program array for a niladic function or is a character array which represents a niladic function.

This function does not exist in *APL\360* but has been the topic of much discussion [Watson 21] . Evaluate is an unusual function in that it may or may not have a result depending upon the expression evaluated. It is specifically defined on program arrays. Then a niladic function D may be defined from a character array G and immediately evaluated by the expression:

$$\perp D \leftarrow \nabla G$$

Recall that the result of specification is its right

operand. If there is no need to assign a name to the function then the following expression is sufficient:

```
⍎∇G
```

Since this is perhaps the most common use of evaluate, the function is extended to include the capabilities of the defined function when it is applied to character arrays. Therefore ⍎*G* is equivalent to ⍎∇*G*.

Since every unheaded defined function is niladic, they are all acceptable to the evaluate function. This is the principle value of an unheaded function.

Examples:

An important use of ⍎ is the creation of names. The following function creates names consisting of the character '*N*' followed by a positive integer and then assigns to the name the value of the integer.

```
      ΔNAMES
[0]   _ N:I
[1]   I←0
[2]  L1:→(N<I←I+1)/0
[3]   ⍎'N',(⍕I),'←I'
[4]   →L1
```

```
      NAMES 20
      N15
15
      N7
7
```

The character vector formed in line 3 is an unheaded defined function having one line.

The evaluate function is sometimes called unquote because of the following identity:

⊥'any expression' ↔ any expression

Thus the evaluate treats a character array like a piece of *APL* notation in that the mention of ⊥ and its quoted operand is much like mentioning an expression without quotes. The symbols ⊥ for evaluate and ⊤ for character form are due to Jim Ryan [Ryan 20].

K. Arrays of Functions

So far the only non-scalar arrays which have been considered were those whose elements had descriptions of type numeric, character, or position. Arrays of program scalars have been limited to rank 0 arrays. There is, however, no reason to require this limitation. A non-scalar array some of whose elements are program scalars is a well defined object. The only question is how these arrays should be evaluated.

Since an array of functions is fundamentally just an array, it would be expected to conform to its operands (if any) in much the same way as operands of a dyadic functions conform with each other when conformability is required. As a simple example consider the vector of functions *W* defined by the expression

```
      W←Δ+,Δ-
      W 5
5 ¯5
      3 W 5
8 ¯2
```

Thus even without a description for evaluating arrays of functions, the intent of the above expressions is clear. (The example also demonstrates that while *W* is an array of scalar functions, it is not itself a scalar function since when it is applied to scalar operands it yeilds a 2 element

vector.) Following are the formal descriptions for arrays of functions on array operands. *M* is any array of monadic functions, *D* is any array of dyadic functions.

1.) Arrays of Monadic Functions *Z* ↔ *M R*

Z ↔

IF Δ*M* is scalar *THEN* *M R*

(the usual definition of *M*)

ELSE IF *E* ↔ ρΔ*M* *THEN*

IF *E* ↔ ρ*R* *THEN* ⊂ *MJ* ⊃*R*

for *MJ*←⊃Δ*M*

ELSE *MJ R* for *MJ*←(ρ*R*)ρΔ*M*

ELSE IF *E* ↔ ρ*R* *THEN* *M* (ρΔ*M*)ρ*R*

ELSE IF ρΔ*M* ↔ ρ*R* *THEN*

Z of dimension ρ*R* such that

I ⌷ *Z* ↔ ⊂ *MI* ⊃*I* ⌷ *R*

for *MI*← ⊃*I* ⌷ Δ*M*

for each PI *I* of *R*

ELSE undefined

Arrays of dextri-monadic functions have a similar description.

Since program scalars cannot be both manipulated and evaluated in a single expression, the names *MI* and *MJ* are specified with the appropriate scalars and these names are applied to the operands.

In the following description of an array of dyadic

functions, the same conformability rules are enforced on a triple of arrays. Since the conformability of the operands does not change from that of a dyadic scalar function, the following simplification can be made. If $E \leftrightarrow \rho L$ make the replacement $L \leftrightarrow (\rho R)\rho L$. If $E \leftrightarrow \rho R$ make the replacement $R \leftrightarrow (\rho L)\rho R$. these replacements allow concentration on the effects of the array of functions and are assumed in the following description at each level of recursion.

2.) Arrays of Dyadic Functions $Z \leftrightarrow L\ D\ R$

$Z \leftrightarrow$

IF ΔD is scalar *THEN* $L\ D\ R$

(the usual definition of D)

ELSE IF $E \leftrightarrow \rho \Delta D$ *THEN*

IF $(E \leftrightarrow \rho L)$ *AND* $(E \leftrightarrow \rho R)$ *THEN* $\subset (\supset L)\ DJ\ \supset R$

for $DJ \leftarrow \supset \Delta D$

ELSE IF $\rho L \leftrightarrow \rho R$ *THEN* $L\ DJ\ R$

for $DJ \leftarrow (\rho L)\rho \Delta D$

ELSE IF $(\rho L \leftrightarrow \rho R)$ *AND* $(\rho L \leftrightarrow \rho \Delta D)$ *THEN* Z of dimension ρL such that

$I \neq Z \leftrightarrow \subset (\supset I \neq L)\ DI\ \supset I \neq R$

for $DI \leftarrow \supset I \neq \Delta D$

for each PI I of L

ELSE undefined

In general, the operands of a non-scalar array of

functions have less freedom of structure. First of all shapes must always conform; and then at each step in the recursion, operands are revealed. It is therefore difficult to have a primitive in an array take a unit array as an operand. However this is a small restriction because the result is concealed.

Examples:

```
      F ← 2 2ρΔ+,Δ-,Δι,Δρ
      F 5
0 0:ρ
5

0 1:ρ
¯5

1 0:ρ5
0 1 2 3 4

1 1:ρ0
```

```
      6  F  2 2⍴⍳4

0 0⍴⍬

6


0 1⍴⍬

5


1 0⍴⍬

0


1 1⍴⍬6

3  3  3  3  3  3
```

To calculate $(R+1)\times R-1$

```
PM←Δ+,Δ-
×/R PM 1
```

The previous discussion includes arrays of defined functions. In particular, arrays of niladic functions may be evaluated in their character form by the evaluate function in the obvious way. Examples of arrays of defined functions are defered till the section on multiple-processes.

While it is possible to define any array of functions, not all such arrays can be evaluated. For example, an array of functions containing both monadic and dyadic functions cannot be evaluated in either the presence or absence of a left operand!

Evaluation of a mixed variable (i.e. one containing both program and non-program arrays) is possible only if the programs are niladic. Then the result of evaluation is an array identical to the variable description except the programs are replaced by their evaluated results.

If every function in an array of functions may have a function index, then the array may have a function index. The rules for general specification govern the mapping of the specified index onto the array of functions.

Examples: let *M*←Δ+/,Δϕ and *R*←(⊂2 2⍴⍳4),⊂⍳3

```
      M[0] R
0⍴2
2  4

1:⍴3
2  1  0

      M[1 0] R
0⍴2
1  5

1:⍴3
2  1  0
```

```
      M[(⍳0 1),0]  R
0:⍴
6

1:⍴3
2  1  0
```

The operators scalar product, inner product, outer product, and reduction apply to arrays of functions without change. For example the outer product on an array of functions may be used to produce an array of operation tables.

```
      AM←⊂+,⊂×
      ⊃(⍳4) ∘.AM ⍳4
0  1  2  3
1  2  3  4
2  3  4  5
3  4  5  6

0  0  0  0
0  1  2  3
0  2  4  6
0  3  6  9
```

Ordinary outer product applies functions to all pairs of elements one from each operand. This concept may also be

applied between a function array and its operand(s). This is called the function outer product and may be monadic, dextri-monadic, or dyadic. A suggested notation for these function is respectively $\underline{L}\ \underline{D}.\circ$; $\underline{D}.\circ\ \underline{R}$; and $\underline{L}\ \underline{D}.\circ\ \underline{R}$ [Edwards 6]. For the dyadic case, scalar conformability is required for the operands.

These functions are not really needed because the scalar product operator on an array of functions produces this same effect.

Examples

```
      PM←Δ+,Δ-
      ⊃ ⍪ PM 2 3
 2   3
¯2  ¯3

      ⊃8 ⍪ PM 2 3
10  11
 6   5
```

Chapter 3

APL Potpourri

A. Introduction

This chapter presents a mixture of topics which may be classified in three general groups. Sections B-E contain discussion and elaboration on topics already defined in chapters 1 and 2. Sections F-I contain new functions and their applications. They are presented separately because they may be accepted or rejected independently of the rest of the notation. Finally, sections J-O present generalizations for consideration which are not actively proposed as a formal part of this work. Some have been proposed elsewhere and are included for completeness, others originate here but are set aside for the stated reasons.

B. Magic Syntax?

It has been stated that any valid *APL* expression may be evaluated unambiguously by the rules given on page 30. Yet at first glance, it appears possible to write expressions in the notation which are impossible to evaluate or which are ambiguous. This section will discuss the impact on the notation of defined functions which are valid in more than one context and will show that every combination of symbols defines either an expression which may be unambiguously evaluated or a statement which is invalid. Two conditions are isolated under which alleged expressions cannot be evaluated. It will be shown that a well formed expression is not necessarily valid and that a specification can alter the syntax of an expression.

For an example of an expression impossible to evaluate let *M* be a function having a result and having a monadic description, and let *X* be a function having a result and having a dextri-monadic description (see page 23). Then how is the expression

(case 1) *M X*

to be evaluated? It appears that each requires evaluation of the other for its operand. As an example of an apparent ambiguity let *F* be a function having a result and having both a niladic and a monadic description, and let *G* be a

function having a result and having both a monadic and a dyadic description. Then how is the expression

(case 2) *F G V*

to be evaluated? Both *F* niladic - *G* dyadic and *F* monadic - *G* monadic appear to be valid combinations.

Dealing with problems of syntax such as these requires a more formal description of what constitutes a legal expression. In the following discussions of syntax, an expression is assumed to be represented in a cannonical form using the symbols (,), [,], *V*, *F*0, *F*1 where *V* is a variable, a literal array, an identifier not having a description, or an array resulting from a function evaluation , *F*0 is a function or a primitive function which does not produce a result, and *F*1 is a function or a primitive function which produces a result. Recall that a variable function whose elements are program scalars is called a function, all others are called variables.

The following set of rules may be used for the synthesis of fully parenthesized expressions in the cannonical form.

Non-terminal symbols

EXP - an expression

E0 - an expression with no value

E1 - an expression with a value

FN0 - a function with no result

FN1 - a function with a result

Terminal Symbols

(,), [,], *V*, *F*0, *F*1 (same as the cannonical form) and

∧ an empty string

Complete parentheses synthesis rules.

The following substitution rules may be used to transform the root symbol (EXP) into expressions in the cannonical form. The rules are presented in the following form:

N→NT

where N is a non-terminal symbol and NT is a combination of terminal and non-terminal symbols which may be substituted for N. The rules are numbered for easy reference. In case several substitution rules involve the same N, alternate NT's are listed and numbered on the same line separated by the symbol |. The symbols →, |, ∧ should not be confused with their usual use in the notation.

(EXP) - the root symbol

(1-3) EXP → (EXP) | E0 | E1

(4-8) E0 → ^ | FN0 | FN0 (E1) | (E1) FN0 | (E1) FN0 (E1)

(9-10) E0 → (E0)(E0) | (E0)

(11-14) E1 → FN1 | FN1 (E1) | (E1) FN1 | (E1) FN1 (E1)

(15-18) E1 → (E0)(E1) | (E1)(E0) | (E1) | *V*

(19-20) FN0 → *F*0 | *F*0[EXP]

(21-22) FN1 → *F*1 | *F*1[EXP]

Any expression derived by these rules and by application of the parentheses elimination rules is called a well formed expression. Appendix 3 gives a set of rules which classify the terminal functions of the generated expression as *N*, *M*, *X*, or *D* for niladic, monadic, dextri-monadic, or dyadic respectively and which include the parentheses elimination rules. The explosion in the number of rules is principly caused by the formalization of the rather heuristic rule which allows parentheses not altering the scope of operands for some function to be deleted.

When these rules are used to determine the classification of an expression, they are used with foreknowledge of what result is desired (i.e. synthesis not analysis). If one applies the wrong rule then he will find no path leading to the desired result. Rules which would classify expressions by analysis are possible and would be useful in an implementation.

A string of symbols selected from the terminal symbols may fail to be a well formed expression in only a few ways:

1.) If it contains unbalanced parentheses or brackets.

2.) If it contains pairs of parentheses and brackets whose ranges intersect.

3.) If it contains a function index bracket divorced from a function symbol. (Brackets are not used for indexing as in *APL\360*.)

4.) If it or any parenthesized part of it defines more than a single array.

This last case is legislated so that expressions may properly be called compound functions. (functions have a single array as a result or no result at all.) Note that the rule only applies to a complete expression or a complete parenthesized sub-expression.

If an expression contians neither a specification nor a defined function which alters global variables, then it is said to contain a syntax error if it is not well formed. If an expression does contain a specification, then some "magic" can occur and the concept of a well formed expression is not so useful. For now consider expressions

which have no embedded specifications.

The following statements fail to be well formed expressions:

V F0 (V by 1

(V F1[V)] by 2

V[] by 3

V V by 4

The following are well formed expressions:

V F1 V

(V F0) F0

F1 F0 V V

This last is well formed despite the occurrence of *V V* because *V V* is neither a complete expression or a complete parenthesized sub-expression. Appendix 3 gives the right-derivation of these expressions and classifies the functions. Evaluation of *F0 V* gives no result leaving *F1 V* remaining for evaluation.

Now the original two cases from page 186 may be examined. Case 1 in cannonical form is

*F*1 *F*1

There are two possible parenthesized expressions from which this one could have been derived:

(*F*1 (*F*1)) and ((*F*1) *F*1)

and therefore the functions may be classified as

(*M*1 (*N*1)) and ((*N*1) *X*1)

Applying the parentheses elimination rules from page 29 gives

*M*1 *N*1 and (*N*1) *X*1

The first of these is then the correct classification of the original expression *M X*. Thus it is not possible in the notation to write a monadic followed by a dextri-monadic, and evaluation is impossible only because neither function has a niladic description. (The same argument holds if *X* is dyadic so this does not involve a departure from *APL\360*.) The expressions have proper syntax but the functions do not have matching descriptions. Any

such statement is said to contain a definition error.

Case 2 in cannonical form is

*F*1 *F*1 *V*

Two possible parenthesized forms are

(*F*1 (*F*1(*V*))) and ((*F*1) *F*1 (*V*))

and therefore the functions may be classified as

(*M*1 (*M*1 (*V*))) and ((*N*1) *D*1 (*V*))

Applying the parentheses elimination rules from page 29 gives

*M*1 *M*1 *V* and (*N*1) *D*1 *V*

The first of these is then the correct classification of the original expression *F G V*. The second is the other possible meaning. Notice that the parentheses which remain in the second expression are needed to specify function precedence other than increasing from left to right. This does involve somewhat of a departure from *APL\360* in that a niladic function as the left operand of a function requires parentheses. These parentheses would be called superfluous

by some [Breed 2] and the resulting syntax ambiguous. Yet can parentheses which define the number or scope of operands for a function be superfluous?

The existence of parenthesized expressions having no value gives rise to another syntatic departure from *APL\360*. In *APL\360*, to say that "the rightmost function whose operands are available ... is evaluated" [Lathwell,Mezei 11] is equivalent to saying that functions are evaluated in order of decreasing precedence (see page 30). This is not true in generalized *APL* and an example is:

(V F0 V) F1 V

Since *F1* appears immediately to the right of a right parenthesis, it has lower precedence than *F0*. Therefore *F0* is evaluated first and gives no result. *F1* is evaluated second and is monadic. Thus *F1* is not evaluated first even though it is rightmost and its operand is available. The classification of this statement appears in Appendix 3.

Given the classification of an expression it is then possible to determine the function of highest precedence and so evaluation of the expression can begin with evaluation of that function. Clearly the next function to be evaluated must be the one of next lower precedence. However this may not be the case (!). This unexpected (and perhaps

disturbing) condition occurs because evaluation of the first function may change the syntax of the statement. This bit of magic may occur if the function in question is a specification or is a defined function which specifies global variables. For example consider the following expressions the last of which is classified:

```
      MEAN←5
      4 MEAN 6,ρMEAN←∇'Z←A _ B ⍞ Z←.5×A+B'
      V  V  V  D1 M1 V  D1 M1 V
```

But this defines three arrays. One from *V D1 M1 V D1 M1 V* and two *V*'s. Therefore this would appear to have a syntax error. Yet during evaluation of the expression *MEAN* is respecified to be of type program and is therefore reclassified as cannonical type *F1*. The expression is then evaluated as though it were classified:

```
      V D1 V D1 M1 V D1 M1 V
```

This expression does not contain a syntax error and may be evaluated.

Syntax errors may also be introduced during evaluation of an expression.

```
REV←Δϕ ⍞ A← 2 2⍴⍳4
REV[0] A ⍪ 0⍴REV←5
```

This statement could be classified

```
F1[V] V D1 V D1 V D1 V
```

This does not contain a syntax error. Yet during evaluation *REV* is respecified to be of type numeric and now the statement contains brackets divorced from a function and a syntax error is introduced.

Thus the apparent syntax of an expression as separated from evaluation is not necessarily the same as the actual syntax during evaluation. Thus, the classification of any expression is valid, in general, only until the first function is evaluated. This is why a right-derivation is used in all the examples. After a specification or a defined function which alters global variables has been evaluated, the remaining expression must then be reclassified. Therefore the syntactic analysis of an expression in the generalized *APL* notation is intimately tied to the evaluation of the expression.

Even without this requirement the dynamic structure of arrays in *APL* would make compilation of expressions (in the usual sense) difficult in that the declaration of array structure and data types may change during evaluation. For

this reason implementations to date have been interpreters (an interpreter alternates between analysis and evaluation where a compiler analysis entire sets of expressions before any evaluation). The generalizations proposed in this paper imply that even the syntax of an expression may change during evaluation and this makes compilation of the notation even more painful if possible at all. This could well mean that the notation is inherently interpretive!

C. Call by Name

The *APL* notation has been designed such that the actual operands of a function are evaluated and the result is used wherever the corresponding formal parameter occurs. This is sometimes referred to as a call by value. (early programming languages referred to the invocation of a function or subroutine as a call.) Therefore in one sense only constant values may be parameters to a function. Yet it is frequently desirable to specify a function or an expression as an actual parameter to a function such that this function or expression is not evaluated upon invocation but rather is evaluated at each occurrence of the corresponding formal parameter. This is usually called a call by name. There are various ways to achieve this in the generalized notation.

First, a function reference or an expression may be passed as a character constant then evaluated where needed by the evaluate function. For example consider a dyadic function *SUM* which adds up terms of a series from *N*=0 to the integer specified as the left operand, where the right operand is the expression for the *N*th of the series (expressed in characters in terms of *N*).

This function does the call by name as advertised but there are some objections to the method. In the first example a constant series is specified (constant *N*th term). It is somewhat painful to pass a numeric in its character form especially if the numeric is the result of some expression rather than literal (of course the character form function could be used). If one had a function and wished to pass to it the character '*C*' by name then the operand would be written ''''*C*''''.

A user of the function must be aware of the local names used in the function and be careful not to use them. For example:

```
TERM←4 ⌷ 3 SUM 'TERM'
```

is invalid while

```
TER←4 ⌷ 3 SUM 'TER'
```

is perfectly valid. This is because *TERM* is local to *SUM* and the local value is used in the evaluation.

A final objection has to do with the function *SUM* itself. The function is designed with the fore-knowledge that call by name using evaluate is to be used. Often in the notation, a function may be designed assuming a scalar operand and yet it works properly for array operands (this is because of the orderly way in which scalar functions are extended to arrays). It would be elegant if a function could be designed expecting a call by value and yet work properly

```
      ΔSUM
[0]   SUM←LAST _ TERM;N
[1]   (SUM,N)←0
[2]  L1:→(LAST<N)/0
[3]   SUM←SUM+⊥TERM
[4]   N←N+1
[5]   →L1
```

Let *F* and *G* have the following definitions:

```
      ΔF
[0]   Z←_
[1]   Z←2+N*2

      ΔG
[0]   Z←_V
[1]   Z←2+V*2

      3 SUM '4'
16
      2 SUM 'F'
11
      2 SUM 'G N'
11
      2 SUM '2+N*2'
11
```

```
      2 SUM1 ∇'2+N*2'

11

      TERM←4 ⌷ 3 SUM1 TERM

16
```

There are several points of interest here. First, the call by name property lies with the actual parameter to the function and not with the formal parameter or with the function itself. Second, the function is (at least supposedly) designed to expect values yet it works properly if given a function or expression which produces a value. In fact if given an array of *N*th terms, the function would evaluate an array of sums. The key is that both define and definition-of evaluate to a program description which is passed and a description is always a constant.

Thus, call by name is really a misnomer for it is not the name which is of interest at all but rather the description associated with the name. The last example shows that the problem with conflict of names is reduced. The following example shows that the problem is not eliminated.

```
      TERM←2 ⌷ 2 SUM1 ∇'TERM+N*2'
```

Here the description passed still involves a name and the local value is active and therefore the expression is invalid. The following expression is worse yet because it is not invalid but gives erroneous results.

```
      SUM←2 ⌷ 2 SUM1 ∇'SUM+N*2'
```

The next example shows that the conflict of names can always be reduced to affect only a single name by using strictly local variables and a secondary function to calculate the *N*th term.

```
      ΔSUM2
[0]   SUM←LAST _ TERM;;N SUM LAST L1
[1]   (SUM,N)←0
[2]   L1:→(LAST<N)/0
[3]   SUM←SUM+ NTH TERM
[4]   N←N+1
[5]   →L1

      ΔNTH
[0]   TERM← _ TERM
[1]   TERM←TERM
```

Here the only possible conflict can occur with the name *TERM*. This scheme also works with evaluate.

Therefore when functions and expressions are not special objects in a language, call by name is not essentially different from call by value.

D. *IF APL* has a Conditional *THEN* !!! *ELSE* ???

Perhaps the greatest shortcoming in the notation is the lack of an explicit conditional function. While much of the meta-notation has been assimilated in the notation, the conditional is conspicuous in its absence because of its frequent use in descriptions. In this section some of the problems associated with the attempt to define such a function are discussed along with some methods for circumventing the problem.

The conditional may be characterized as follows:

1.) It is a triadic function (antecedent, consequent, and alternative).

2.) Evaluation of the antecedent determines which of the other operands is evaluated.

The fundamental problems in attempting to define a conditional function in the notation are:

1.) The syntax does not permit triadic functions. Supplying a special syntactic type for this one function would not be an elegant solution.

2.) Operands to a function are always evaluated before the function itself is evaluated. But in the conditional one operand is by definition not evaluated and need not even be a well formed expression.

This does not mean that the effect of a conditional function is lost to the notation for it may be acheived in many ways some of which shall be described in the pages that follow.

Evaluate and the Conditional

Evaluate may be used to gain the effect of the conditional. The statement

```
IF X THEN Y ELSE Z
```

might be realized by the expression

```
⊥ X ⍎ ∇'Z')∇'Y'
```

This formulation uses indexing to select from a two element vector of expressions. The vector may arise from define (∇) applied to a character operand (as is done here) or from definition-of (Δ) on a defined function. The unselected expression is never evaluated and need not be a valid expression.

The definition of Interval from page 43 could be realized by the following function.

```
      ΔINTERVAL
[0]   I←_R
[1]   I←⊥(R=0) ⍎ ∇'(_R-1),R-1-?1',∇'⍳0'
```

(*Z*←→*?R* is a scalar function and produces a random integer *ORG*≤*Z*<*ORG*+*R* hence *?*1 is the index origin.)

When conditionals are nested, one is faced with writting very long lines containing quoted expressions which themselves contain quoted expressions to many levels. Using the character form of an expression is not appealing in any case.

Branching and the Conditional

The statement

IF X THEN Y ELSE Z

may be realized in a defined function by using branching as follows:

```
→X↓L1 ⍬ C←Y ⍬ →L2 ⍬ L1:C←Z ⍬ L2:etc.
```

where *C* is the value of the conditional

The first branch calculates either to an empty vector or to *L1*. If *X* is true (=1) then *Y* is evaluated and the second branch skips evaluation of *Z*. If *X* is false (=0) then evaluation of *Y* and the second branch is skipped and *Z* is evaluated. In either case line *L2* is the next evaluated.

Since branching is a function without a result and since

```
(E0) E1 (E0)
```

is a well formed expression, a shorthand version of the above formulation can be given as

```
(→L2) C←Y (→X↓L1) ⍬ L1:C←Z ⍬ L2:etc.
```

This is little better than the original, but shows a use of an embedded branch. In this expression, *X* false means quit expression evaluation early. Thus the total expression need not be well formed yet no error is detected.

The definition of interval could be realized by the following function:

```
      ΔINTERVAL
[0]   I←_ R
[1]   →(R=0)↓L1 ⍝ I←⍳0 ⍝ →0 ⍝ L1:I←(_ R-1),R-1-?1
```

(This function has 5 lines counting the header for writing several expressions on a single line is merely a topograpgic consideration.)

Nesting of conditionals is moreconvenienent than with evaluate. For example, the definition of Vector Take from page 45 could be realized by the following function:

```
      ΔTAKE
[0]   Z←L _ R
[1]   →(L≥0)↓L1
[2]   (→0)Z←(⍳L)⌷ ,R(→(≤⍴,R)↓L11) ⍝ L11:(→0)Z←R,(L-⍴,R)⍴0
[4]   L1:→(L<0)↓L2
[5]   →(L≥-⍴,R)↓L21
[6]   (→0) Z←((⍳|L)+(⍴,R)-L) ⌷ R
[7]   L21:(→0) ((|L+⍴,R)⍴0 ),R
[8]   L2:'UNDEFINED'
```

The use of branching to acheive the conditional has some interesting properties. First the labels used make it easy to tell which *IF* goes with which *ELSE*. Second, statements may be combined in non-standard ways. (e.g. the *ELSE* clause of one conditional may be the *THEN* clause of another.)

Thus it has been shown that the inability to define triadic functions and the evaluation rules for operands to functions do not exclude the functional abilities of the conditional from the notation.

E. Recursive Functions

The term recursive function refers to a function which is defined in terms of itself and can therefore only be properly applied to a defined function. Either an unheaded or a headed defined function may be used to define a recursive function. A good example of recursion is the factorial function. It is such a common example that no one is concerned with the factiorial so there is nothing left to look at but the techniques! The following is an unheaded recursive function for factorial *N*.

```
      ΔFACT1
[1]   →(N=0 1)/0
[2]   F←F×N
[3]   N←N-1
[4]   FACT1
```

This function is recursive because on line 4 it refers to itself. Since it is unheaded its operands are implied and must be set prior to evaluation. In this case the variable *N* is the operand and the variable *F* leaves the result variable and must be initialized to 1 externally to the function. It cannot be initialized inside the function because the global nature of *N* implies that the partial factorial be computed before recursion. The last line evaluated is line 1 which is a branch and therefore by convention the function has no

result. So an extra expression is required to display the result.

Example:

```
      F←1 ⍞ N←4
      FACT1 ⍞ F
24
```

*FACT*1 generally lacks any similarity with the ordinary definition of the factorial.

The headed version is far superior in this case (and in almost every case).

```
      ∆FACT2
[0]   F←N _
[1]   F←1
[2]   →(N=0 1)/0
[3]   F←N×(N-1) _
```

Notice that the function is dextri-monadic and matches the usual mathematical syntax for the factorial (which is no great advantage). Line 3 is a translation of the expression $N! \leftrightarrow N\times(N-1)!$ The fact that *F* and *N* are local means they do not collide with *F* and *N* of the recursive call so the result variable may be initialized within the function. The recursion is denoted by the placeholder and so is

independent of the function name.

Example:

```
      4 FACT2
24
```

Neither function will terminate if the operand is other than a non-negative integer.

There is a subtle difference between these functions. Suppose the following expressions are evaluated:

```
Q1←ΔFACT1
Q2←ΔFACT2
```

Then *Q*1 is a function having the same description as *FACT*1 and therefore they are the same function. A similar statement is true for *Q*2 and *FACT*2. Yet *Q*1 is not recursive while the others are! This is because *Q*1 does not refer to itself in its description (it still refers to *FACT*1). Therefore any recursive property of an unheaded defined function must be considered transient. When the placeholder symbol is used to specify recursion in a headed function then the recursion is a permanent property of the function and is preserved when the program scalar is given a new name.

F. Specification in the Name Domain

In Chapter 2 arrays of names were used to define a selective form of specification. At no time, however, was a name array itself associated with a name. There appears to be no reason to continue this embargo. Therefore a specification is considered which has the same syntax as value specification ($Z \leftrightarrow L \leftarrow R$) but which has both operands in the name domain. At the moment a right operand of a function is in the name domain only if it is already part of an expression left of specification. Its definition is the same as simple specification except that R is an array of names. The extension to general specification is avoided so no array contains both name scalars and other scalars. The effect of the function is to permit naming of subarrays of an array.

Example:

```
      V←⍳10 ⋄ (W←5↓8↑V)←-⍳3
      V
0  1  2  3  4  0  ¯1  ¯2  8  9
```

The above example has defined W to name a subarray of V. The description of W is a three element vector of name scalars. The mention of W in the value domain implies an indirect reference to the values named.

```
      W
0  ¯1  ¯2
```

The value specification in this example is a trick to get the expression (*W*←5↓8↑*V*) into the name domain but alters *V* as though the name specification were not written. This is because the name specification has its right operand as its result. A further refinement of names introduced in the next section along with a new function will remove the need for the value specification.

The treatment of names (as opposed to name scalars) in the name domain is modified such that mentioning the name of a name array is equivalent to mentioning the array itself. This is required so that

```
      W←'ABC'
```

remains an indirect reference to *V* rather than a description replacement.

A name array fails to act like an ordinary variable in two ways. First, when it is the left operand of a specification, it has stronger conformability requirements. Using the previous *W*, note that when *W* is respecified, the right operand must conform to the dimension of *W* or be a unit array.

```
      W←0
      V
0 1 2 3 4 0 0 0 8 9
      W
0 0 0
```

Second, a named name array is dependent upon the rank and, to a lesser extent, the dimension of another variable. If *V* is respecified to be other than rank 1, or less than dimension 8, then both attempted reference and specification of *W* are invalid.

Examples

```
      N←2 2⍴⍳4 ⋄ (DIAG←0 0⌹N)←9 10
      N
9  1
2 10
      DIAG
9  10
```

but `N←⍳10 ⋄ DIAG` is invalid.

Specification in the name domain adds a new breadth of naming conventions to the notation and careful consideration of its impact will be required. The fact that it is strongly rank and dimension dependent is unfortunate. It would be useful, for example, to name the diagonal of a rank 2 array independently of its dimensions. Since the name array essentially indexes the referenced arrays, it is possible that a solution to the above problem is tied to a more general indexing function.

G. Active and Passive Expressions

When a variable function used as an operand is evaluated, the resulting expression is normally viewed as being detached from the name of the variable. Such an expression is called a passive expression in that its use as an operand of some function can never affect the description of the originating name of the expression. Any expression written in the notation or produced by functions introduced thus far are passive. An operand of a function is called active when it is altered by the function evaluation. An active expression is used in the notation to describe an indivisibility of function evaluation and specification. The description of an active operand to a function becomes the result of the function evaluation. The evaluation and specification are considered a single action.

Activate (A)

Syntax: $\underline{Z} \leftrightarrow \underline{L}\leftarrow$

$\underline{Z}$ is the active expression of $\underline{L}$

Conformability: $\underline{L}$ is any expression valid in the name domain.

Activate is the only dextri-monadic primitive defined. This syntax is chosen to suggest a similarity with specification. The display convention also reflects this similarity in that display of a result is suppressed in case it is the result of the activate function or any function one of whose operands is activated.

The function is used for two widely differing applications. First it provides an easy way to denote expressions in the name domain and therefore may be used for subarray naming. The name arrays from the previous section may now be generated without respecifying any values:

```
(W←5↓8↑V)←
(DIAG←0 0⍉N)←
```

For these examples the fact that the result is active is not relevant.

The following examples show active expressions used as operands:

```
A←5
1+A←      (the successor function)
```

```
      A
6

      V←⍳10
      (V←),5 6
      V
5 6

      A←B←5
      (A←)+B←
      A,B
10  10
```

Activate has been used above to avoid writing a name twice. Another scheme proposed for doing this is $A+\leftarrow 1$ to mean $A\leftarrow A+1$ [Ryan 20]. An active expression for $5\div A$ would not be possible in that notation. Using activate both $5\div A\leftarrow$ and $(A\leftarrow)\div 5$ are valid and do not require a special syntax to explain a monadic looking specification.

In case both operands of an active expression are the same, the order or number of activations is not important for two reasons. First the operands are evaluated before the function and second there is at most one result of the function. Thus the following expressions are equivalent:

```
B+B←
(B←)+B
(B←)+B←
```

If an active variable is used as operand to a function having no result, then the variable name is detached from its description. In a sense the value of the variable is erased. The expression

```
→A←
```

will erase *A* (because → has no result) but has the unfortunate side effect of branching. A new primitive could be defined but the following defined function is sufficient.

```
    ΔERASE
[0]   _ R
```

Then the expression

```
ERASE A←
```

will erase *A*.

None of the examples given here are exciting uses of activate. The real significance of active expressions will be made clear in the following section on multiple functions.

The denotation for activate does not conflict with the similar notation used in a function header, because a function header is not an expression.

II. Multiple Functions

The array orientation of *APL* notation implies a parallelism of actions. For example in the evaluation of the statement

```
(2 3⍴⍳6)+2 3⍴⍳6
```

the implication is that six additions are done simultaneously producing six results. In a digital computer the computation may indeed be done serially but this is transparent to the notation. Parallel computers have been designed and it would seem that *APL* may be a useful language for them.

A well formed expression may be invalid only because the parallelism defines a race condition. For example the expression

```
(0 1 0⌹A)←⍳3
```

must be considered invalid because the value of `0⌹A` at its completion is ambiguous. In an implementation on a serial computer (like *APL\360*) this condition is expensive to detect and an invalid expression as this may in fact be evaluated. Knowledge of the implementation (or a little experimentation) would allow one to predict the value of `0⌹A`. However use of such an implementation shortcoming could produce different results on different *APL* systems.

The real implications of parallelism are not striking till arrays of defined functions are considered. If the defined functions (and any functions referenced by them)

have no common implied operands, (i.e. global variables) whose values are respecified, then they are not essentially different from primitive functions and evaluation of each function is independent of the others. Although conceptually the functions are evaluated simultaneously, there would be no way to tell in an implementation if this was in fact the case. If, on the other hand, two functions in an array mention a common variable and if at least one of the functions respecifies the variable, then evaluation of the array may be ambiguous. This ambiguity is in general difficult to detect.

For example consider the array of unheaded defined functions defined from the character array G:

```
      G
(0 ⍪ A)←0
(1 ⍪ A)←1
(0 ⍪ A)←2
      ⍎⊥∘.[1]G
```

First note that the expression does indeed indicate evaluation of an array of functions. The conceal hides each row of G; the scalar product operator applies evaluate (⊥) to each hidden row simultaneously. This is not the same as

```
      ⊥G
```

which causes evaluation of the rows in sequence (the latter expression is even valid). The function array evaluation is

ambiguous for exactly the same reason as

```
(0 1 0 ⌸ A)←⍳3
```

The two expressions `(0 1 0 ⌸ A)←2 3` and `⍸⍳⊂[1]G` are essentially equivalent and demonstrate in a dramatic way the hidden parallelism of *APL*.

If any meaning is to be given to such arrays of functions, there must be a notation provided to state any dependencies existing among parallel functions which makes their synchronization explicit and unambiguous. Such synchronized functions are sometimes called a system of functions.

Conceptually each function in an array is advancing in its evaluation simultaneously with and independently of each of the other functions in the array. Synchronization may be achieved by permitting a function to delay conditionally at some point in its evaluation.

Wait (A)

Syntax: $\underline{Z} \leftrightarrow \underline{L}\omega\underline{R}$

$\underline{Z}$ is $\underline{L}$

Conformability: $\underline{L}$ and $\underline{R}$ are any arrays.

The evaluation of this function may be described as follows. If $\underline{L}$ and $\underline{R}$ are different arrays then evaluation of the function is delayed, if $\underline{R}$ and $\underline{L}$ are the same array then the value of the function is this common value.

The wait function may be used to cause one function to remain dormant till another has computed some required result. Function 1 waits during evaluation of the expression

```
A←0 ⍬ →(1ωA)/L1
```

Function 2 causes function 1 to resume evaluation by evaluating the expression

```
A←1
```

in which case function 1 branches to $L1$. The wait function implies a calculation to test equality every time $\underline{R}$ is respecified.

Use of active operands may cause an implied wait. If function 1 is evaluating the expression

```
1+A←
```

then if function 2 evaluates

```
2+A
```

its evaluation is delayed till function 1 completes its active use of *A*. The inseparability of the function evaluation and the respecification of *A* implies that a parallel reference or specification of *A* must be pendant. The wait is implied over evaluation of a single function and so muy seem a bit restrictive. However the following section shows that this is no restriction at all.

Example Synchronized Multiple Processes

The following example is a modification of a model proposed by Dijkstra [Dijkstra 5 pg.53] and consists of *N* defined functions which are looping as described in figure 5. Each function has a section which may run in parallel with any of the other functions. Each also has a critical section and the constraint imposed is that only one function may be in its critical section at any given time.

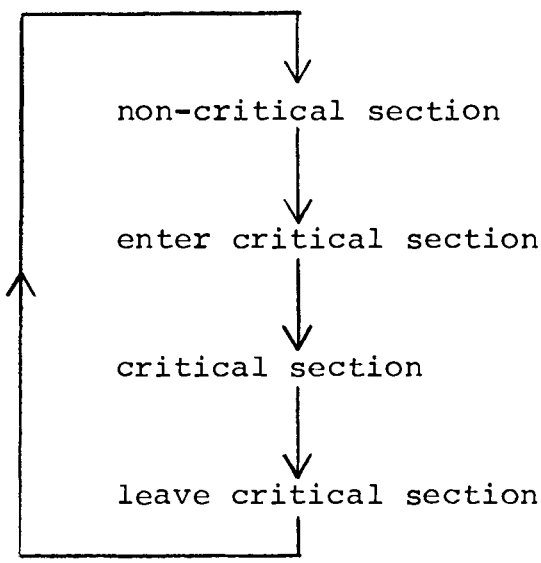


Figure 5 Looping Defined Functions

Dijkstra's solution involved introduction of synchronizing primitives [Dijkstra 5 pg.67] which combined inseparable function evaluation and specification of special objects called semiphores, with delay of evaluation. Here this combination is avoided. The delay is embodied in the

wait function (ω), with the other properties existing as active expressions (see Activate). No declarations or special objects are used.

The following pseudo-function is a prototype for the *N* defined functions.

```
      ΔPROCESSi
[0]   _
[1]   L1:NONCRITICALi
[2]   ENTER i
[3]   CRITICALi
[4]   i LEAVE Q←
[5]   →L1
```

A real function is derived by substituting a positive integer $\leq N$ for each occurance of i in the pseudo-function. To avoid confusion in the following discussion, the functions *PROCESS*i shall be termed processes and the functions they reference shall be termed functions. The program vector *PROCESS* is defined to be a length *N* vector of these processes (i.e. evaluating *PROCESS* implies evaluating the processes in parallel). The actual computation done in the critical and non-critical sections is not of interest here and shall not be specified.

The actual synchronization is done by the *ENTER* and *LEAVE* functions. There is only one copy of each of these functions and the integer i is passed as an operand. Of

course one function may be invoked several times at once by several processes (as with + in array addition). The controlling functions are defined as follows:

```
      ΔENTER
[0]   _ I
[1]   (Q←),I
[2]   IωFIRST

      ΔLEAVE
[0]   Q←I _ Q
[1]   Q←1↓Q
```

The parallel process is initiated by evaluating:

```
Q←,1 ⋄ (FIRST←1↑Q)← ⋄ Q←⍳0 ⋄ PROCESS
```

The global variable *Q* is a queue and is the vector of process numbers desiring use of their critical sections. *Q* empty (as it is initially) means no process is in its critical section. *Q* non-empty means process 1↑*Q* is in its critical section and the processes 1↓*Q* (if any) are waiting to enter their critical sections.

The *ENTER* function places a process on the queue. Notice that *Q* is used actively on line 1 and so the catenation and the respecification of *Q* are inseparable. This is not equivalent to *Q*←*Q*,*I* for using the latter two processes could then define the following sequence:

process number	action
1	evaluate *Q*,1 yielding expression *EXP*1
2	evaluate *Q*,2 yielding expression *EXP*2
3	respecify *Q* as *EXP*1
4	respecify *Q* as *EXP*2

Process 1 would then be lost from the queue.

After a process is entered on the queue, the function waits till the process number is first on the queue. *FIRST* is a global variable defined as a subarray name for the first element of *Q*. Thus the processes are encountered on a first come first serve basis and no process will ever be locked out.

The function *LEAVE* merely deletes the process number from the queue. Notice that the expression $Q\leftarrow 1\downarrow Q$ is used. Since only one process can be in its critical section at one time, only one process can evaluate *LEAVE* at one time. However an *ENTER* and a *LEAVE* could be evaluated at one time. Why then is there not danger of a race condition as before?

The answer is simple but has far reaching implications. When evaluation of *LEAVE* is requested by *PROCESS*i, the operand *Q* is used actively. Therefore any attempt to reference or specify *Q* by another process is pendant over the entire evaluation of the function *LEAVE*. This is why the wait implied by active use of an operand for a single function is not a restriction. The single function may be a

defined function of any complexity.

This simple case could in fact have been written such that each critical section used an active operand to block the others. While this would meet all the requirements of the problem, the order in which processes entered their critical sections would be unspecified (although first come first serve would be a reasonable choice for implementation).

If the processes described here were computing systems, then the critical sections might include access to an I/O channel shared among them.

Changing the *ENTER* and *LEAVE* functions can change the entire personality of the system. For example using the following *ENTER* and *LEAVE* functions a system is defined where $M \leq N$ processes are permitted in their critical sections at the same time.

```
      ΔENTER
[0]   _ I
[1]   →(M≥ρ(Q←),I)/0
[2]   IωFIRST

      ΔLEAVE
[0]   Q←I _ Q
[1]   →(M>ρQ←I~Q)/0
[2]   FIRSTM←¯1ϕFIRSTM
```

This parallel process is initiated by evaluating

```
Q←⍳M ◇ (FIRST←1↑Q)← ◇ (FIRSTM←M↑Q)← ◇ Q←⍳0
PROCESS
```

Here if fewer than M processes are in their critical sections, a new one may enter, otherwise a new process waits till it is first. When a process leaves its critical section, if there are more than M in queue then a new process is granted enterance by rotating its number into the first position of Q by using the subarray name *FIRSTM*.

Again if the processes were computing systems, M might

represent the number of I/O channels shared by them.

A priority system could be defined by having *ENTER* sort the waiting processes according to priority. In general the full power of the notation is available to specify any special requirements a system may have whatever their complexity.

I. Filling in the Void

In this section another data type called void is proposed. It is presented separately because its value to the notation is independent of the other topics in this paper. A void scalar has no literal existence but is produced in two ways:

1.) on attempted evaluation in the value domain of an identifier having no description. (a value error in *APL\360*)

2.) on evaluation of a defined function having a result if that result is never specified. (sometimes a value error in *APL\360*)

In the generalized notation a statement is said to contain a value error in either of the following cases:

1.) the void scalar occurs as operand to a function which does not have void in its domain.

2.) the value of an expression after complete evaluation is void.

This second case means that a display of void is always

an error.

The obvious question is to ask what functions shall be extended to include void in their domains. The answer is easy; none of them. That is, void used as an operand to any function introduced so far implies a value error. However two new functions are defined which do include void in their domains.

Exist (A)

Syntax: *Z* ← ∃ *R*

Conformability: *R* is any array.

Formal description: origin free

```
Z ↔
IF R is void THEN 0
ELSE 1
```

Default-of (A)

Syntax: *Z* ↔ *L* ∃ *R*

Conformability: *L* and *R* are any arrays.

Formal description: origin free

```
Z ↔
IF R is void THEN L
ELSE R
```

These functions make it possible to test if an identifier has a description or if a function defined to have a result, really produces a result. More importantly they provide a more general means for defaulting parameters of a defined function. (see page 164)

The following defined function uses the function header to default its left operand:

```
      ΔMINUS
[0]   Z← (A←0) _ B
[1]   Z←A-B
```

Whenever this function is invoked without a left operand, the expression $A\leftarrow 0$ is evaluated.

Using default-of this could be written

```
      ΔMINUS1
[0]   Z ← (A←) _ B
[1]   Z←(0 ⍎ A) - B
```

Whenever this function is invoked without a left operand the formal parameter is left without a description. (Recall that a formal parameter written in the function header in the form $\underline{L}\leftarrow$ implies that the actual parameter is optional and is not the activation of that parameter.)

This is an important distinction. If one tried to write a times/signum function, one would be at a loss to choose

the proper default value to put in the header because any number is a valid left operand of multiplication. However using exist it could be written:

```
      ΔTIMES
[0]   Z ← (A←) _ B
[1]   →(∃ A)↓L1
[2]   (→0) Z←A×B
[3]  L1:Z← (0<B)-0>B
```

J. Scalar Extension

A number of schemes have been proposed for generalizing the conformability requirements for scalar dyadic functions. The question arises as a practical matter from problems encountered in actual programming in *APL\360*. First, there is confusion about the (unpublished?) rules for conformability of a one element array with other arrays. This is allowed in *APL\360* because so often one element vectors occur where scalars are desired. While convenient in an implementation, such an extension has not been allowed in this work. Second, there are circumstances where it is clear what non-conformable operands to a scalar function ought to mean. A common example is applying a vector to each row or column of a matrix.

One of the earliest published attempts to provide a solution came from S. Charmonman [Charmonman 4]. His proposal (which shall not be restated) solves the above stated problem but suffers some serious drawbacks. It tends to ignore the shape of operands and the shape of the result is somewhat arbitrary; commutative scalar functions do not commute under the extension.

Abrams [Abrams 1] proved the following theorem concerning arrays $\underline{L}$ and $\underline{R}$ conformable for scalar function $\underline{D}$:

$$\underline{L}\ \underline{D}\ \underline{R} \leftrightarrow ((\iota\rho\rho\underline{L}),\iota\rho\rho\underline{R})\circledcirc\ \underline{L}\ \circ.\underline{D}\ \underline{R}$$

By extension one could use the expression on the right as a definition for scalar functions. This allows applying vectors to columns of an array but not to rows (a slight modification allows rows but not columns). A problem in common with Charmonman's proposal is that everything is conformable making errors difficult to detect.

The following proposal [Breed 3] is a compromise which relaxes conformability requirements without removing them. Following is his formal definition for $A \underline{D} B$ for $\underline{D}$ a dyadic scalar function. The formulas are rewritten below in the notation of this paper.

1. $M \leftarrow ((0\lceil(\rho\rho B)-\rho\rho A)\rho 1),\rho A$

   $N \leftarrow ((0\lceil(\rho\rho A)-\rho\rho B)\rho 1),\rho B$

2. $U \leftarrow ((\iota\rho M)+(M=1)\times\rho M)$ ⫤ M,N

   $V \leftarrow ((\iota\rho N)+(N=1)\times\rho N)$ ⫤ N,M

   length error unless $U \equiv V$

3. $X \leftarrow (⍲M=1)⍉((⍲M=1)$ ⫤ $U)\rho(⍋⍲1=\rho A)⍉A$

   $Y \leftarrow (⍲N=1)⍉((⍲N=1)$ ⫤ $V)\rho(⍋⍲1=\rho B)⍉B$

   $R \leftarrow X \underline{D} Y$

   where scalar extension as defined holds.

Simply this means that the operand of smaller rank has its dimension vector extended on the left with 1's. The resulting arrays conform if the dimension vectors are

elementwise equal or one is a 1. Finally each dimension equal to 1 is extended to match the dimension of the other operand causing a replication of the first array along that coordinate.

This scheme has the following features:

1.) It is a superset of the current scalar extension rules.

2.) Commutativity and associativity are preserved where they exist between scalars.

3.) Outer product may be defined in terms of scalar extension.

$A \circ.\underline{D}\ B \leftrightarrow (((\rho A),(\rho\rho B)\rho 1)\rho A)\underline{D}(((\rho\rho A)\rho 1),\rho B)\rho B$

This proposal appears to embody all the requirements of a generalized scalar extension; using it the vector inner product description (page 94) could be written simply $\underline{D}/\underline{L}\ \underline{D}'\ \underline{R}$; its application to arrays of functions does not seem to be a problem; and it is formally pleasing to see the closed outer product definition.

K. Interval Revisited

Interval has already been extended to vectors of non-negative integers. The observation was made that ⍳*R* counts in a base *R* number system. This statement suggests yet a further extension. Since counting in a negative number base is well defined, it should be possible to define interval for vectors containing negative integers. One possible definition for scalar integers is:

```
Z ←→
IF R=0 THEN E
ELSE (⍳R-×R),R+ORG-1
```

The extension to arrays would use this definition for scalars in the description on page 100. This description is obviously the same as before on positive integers.

Examples:

```
      ⍳¯10
0 ¯1 ¯2 ¯3 ¯4 ¯5 ¯6 ¯7 ¯8 ¯9
```

Thus ⍳¯10 counts negatively from 0 to ¯9. Under this definition for non-negative integer *N* ⍳-*N* in origin *ORG* is the same as ⌽⍳*N* in origin *ORG*-*N*

Interval on arrays is even further removed from counting as seen by the following example.

```
      ⍳¯2  ¯2
 0   0
¯1  ¯1


 0  ¯1
 0  ¯1
```

The representation for a number is along the leftmost coordinate. A base value shows which numbers are represented.

```
      ¯2 ¯2 ⊥ ⍳¯2 ¯2
 0  ¯1
 2   1
```

This illustrates the fact that negative radix number systems represent a set of integers skewed about zero.

Thus this extension does not behave as hoped. It may still be considered for inclusion in the notation in its own right. It does produce the mixed radix representation of a dense set of integers.

L. Operand Reversal

Often in an expression, some parentheses could be eliminated if the left and right operands of a function were reversed. There are numerous examples of this phenomena in this paper. For example the parentheses in the definition of vector scan

$$\underline{D}/(I+1)\uparrow\underline{R}$$

could be eliminated if the operands of take were reversed. It has been proposed [Liu 14] that an operator be provided to do this. Choosing ; (only for the moment) as the operator, the above expression could be written

$$\underline{D}/\underline{R};\uparrow I+1$$

But very little is gained. The expression is one character shorter and this is about the best to be expected. The statement is no clearer and is perhaps even harder to read. Therefore this extension is rejected. Parentheses (as any LISP user will agree) are not really so bad.

M. Vacant

Abrams [Abrams 1] proposed that the intent of an expression and not the literal expression itself should be evaluated. Thus

```
0 ⌷ 5 2÷1 0
```

has the undisputed value 5. This view is rather forced since the *APL* machine which he proposes will distribute the select function and will indeed evaluate the expression. However the intent of any expression is subject to interpretation (no pun intended) where the literal expression is not. In the notation as defined here, the expression

```
0 ⌷ 5 2÷1 0
```

must be considered invalid. The question is then what could be added to the notation to make such an expression legal?

Earlier the concept of a value error was formally introduced into the notation under the guise of a new data type called void. It is tempting to try to do the same with other forms of errors. Therefore a domain error scalar called vacant and denoted v is postulated. Vacant is the result of a scalar function on operands not in the domain of that scalar function. As with void, display or specification of an array containing vacant is always invalid. Admiting vacant into the domain of the select functions solves many of the problems. Thus `5 2÷1 0` results in `5 v` and `0 ⌷ 5 2÷1 0` is `5`, while `1 ⌷ 5 2÷1 0` is a domain error. It

appears that vacant ought to be in the domain of the scalar functions so that

```
0 ⍳ 0 3 + 5 2 ÷ 1 0
```

is still defined.

The main objection to this extension is that the danger is present that detection of an error will occur far from its cause, making the cause of the error difficult to determine.

N. Primitive Variable Functions

Variable functions have been divided into two groups: defined functions which are named by identifiers and primitive functions which are named by function symbols. A primitive function is treated much like a constant because no way is provided to change its description. But this changelessness is more a matter of taste than a requirement. Allowing function symbols to be respecified with new descriptions would make them variable in a truer sense. This is specifically rejected for two reasons. First, the standardization of symbols is a practical necessity for a widely used notation. The functional ability of the notation is enhanced by use of commonly distributed defined functions. Second, if it really is desirable to interpret a symbol at one time as + and at another as ∨ then a defined function is suitable when defined once as

```
SYM←Δ+
```

and again as

```
SYM←Δ∨
```

O. Expression Separator Functions

In chapter 2 a new punctuation symbol ⌷ was introduced as an expression separator. It is interesting to note that the symbol could be treated as four functions one with each of the four operand contexts [Rubin 19]. The four functions may be described as follows:

1.) A right operand (if any) is ignored

2.) A left operand (if any) is returned as a result. Only a function with a left operand has a result.

Examples:

```
A+3 ⌷ A←5+6          (⌷ dyadic with result 3)
14

Z←Z*.5 ⌷ →(A<0)/0    (⌷ dextri-monadic with result .5)
1,A← ⌷ →(A<0)/0      (⌷ niladic with no result)
```

These functions may be characterized as follows:

1.) They do not separate expressions but rather join expressions to make a single new expression.

2.) Evaluation of the original expressions

proceeds from right to left.

3.) Display of results in intermediate expressions is suppressed.

4.) Intermediate expressions may be neither labeled nor branched to.

The functions [] differ from the punctuation [] in each of the ways listed above. The fact that the functions provide a single expression is not in itself objectionable. The different order of execution is not traumatic since the order chosen for [] punctuation is somewhat arbitrary. Inhibiting the display of results implies that the functions do not genuinely mimic the punctuation. The functions could be designed so that a right operand would imply a display but this would also imply a display of the result of a specification.

The original purpose of [] was to emphasize the relationship between expressions yet keep them separate for evaluation. Making [] a function defeats this purpose. Thus, the proposal is rejected primarily because of the branching and labeling restrictions it would impose. Some of the more interesting uses of the symbol (the conditional) become invalid.

Chapter 4

Conclusion

A. Summary

The generalizations to *APL* proposed in this dissertation fall into four classifications:

1.) Syntax

2.) Arrays

3.) Names

4.) Functions

The syntax is generalized to permit functions in four operand contexts to exist for primitive functions and defined functions. An expression or any complete sub-expression of an expression is treated much like a function in that it evaluates either to a single array or to no result at all.

Arrays are extended by allowing more scalar types and by defining arrays each of whose elements may be an array of any rank.

The existence of names (in the name domain) as separated from arrays (in the value domain) is recognized. Functions defined on names are generally insensitive to arrays associated with the names.

Functions are defined to create and manipulate new scalar types and general arrays. Functions previously defined on vectors are extended to arrays by use of general arrays for array indexing. A defined function is treated much like a primitive and may describe any of four functions by use of default operands. A function index is permitted with a defined function and in general may be a vector.

For the purpose of this work *APL\360* has been taken to be a de facto standard for *APL*. A detailed summary of the changes to *APL\360* which are proposed is presented in Appendix 1.

The proposals which have been put forward here may be adopted in whole or in part although there are dependencies. These suggestions have been made with the prime focus on the language and not on its implementation; moreover, there are still areas for futher consideration in the language in addition to what is discussed in this paper. At this time it is appropriate to comment on implementation and further work.

B. Implementation

In keeping with the spirit of *APL* the generalizations are presented as a notation not as an implemented programming language. Problems of implementation have been generally ignored. However it is the feeling of the author that all the concepts in chapters 1 and 2 may be implemented in a straightforward manner. Certainly the uniformity of the syntax should ease syntax analysis.

Implementation of functions on general arrays would propably mimic quite closely the descriptions given for them here. In particular the scalar functions should be amenable to a simple recursive evaluation algorithm.

An easy way to represent a general array on a conventional computer would be to store in row major order a set of pointers (addresses) to the arrays in each position of the general array. Each indicated array would either be one of the scalar types or another array of pointers. In case all the scalars of a simple array were of the same type, a compact representation of the array (as with all arrays of *APL*\360) would provide storage optimization. The use of descriptors separated from the values as proposed by Abrams [Abrams 1] is clearly indicated.

The treatment of program scalars has not implied constraints on what the computer representation of expressions should be so long as a character representation equivalent to the original description is always obtainable.

Proper implementation of arrays of functions merely implies multiprogramming of functions. Inclusion of the synchronizing primitives of chapter 3 (activate and wait) may be more difficult.

Implementation of named name arrays (i.e. specification in the name domain) should probably be pendant on further study of the implications of the concept.

C. Further Research

Certain aspects of the generalized notation seem to beg for further extension and a number of problems have not been considered at all.

The position scalar

Prominent in this set is the position scalar ⍬ . It is in the domain of the structure and select functions and very few others. It arises because arrays containing scalars of different type are permitted. In some applications it could be useful if ⍬ were treated as a universal identity for the scalar functions. That is ⍬ in a operand of + would act like a zero, while ⍬ in an operand of × would act like a one. Then an expression like

```
1+1 0 1\3 4
```

would be valid as it is in *APL\360*. Unfortunately like the universal solvent that can't be stored, the p-scalar would be difficult to detect. Neither = nor ≡ (which is defined in terms of =) could detect it. Worse ≡ would not really be identically equal since every occurrance of ⍬ would be treated as a one.

Unit indexing

A more tenable use of ⍬ would be to postulate the following identity for unit indexing

⍬ ⌷ $\underline{R}$ ↔ ⍬

then

```
1 3 0 ⫤ 'ABCDE' ↔ BD 0
```

and

```
(B⍳A) ⫤ B
```

would be an array of the elements of *A* which occur in *B* with 0 elsewhere.

Unit indexing itself seems to need a further extension. While it is a replacement for the bracket indexing of *APL\360*, it is not an equivalent replacement. There is no way to mimic the elision of a coordinate which in *APL\360* implies the selection of all elements along that coordinate. Indexed unit indexing does select all elements along unindexed coordinates but the resulting array is short by a transpose of being identical to the *APL\360* notation. In the absence of such an extension, the bracket notation for indexing could be retained in an implementation solely for convenience even though ⫤ is more pleasing formally.

A different extension to unit indexing would permit selecting of elements on other than the top level of an array.

Functions on General Arrays

Only a few new functions have been defined on general arrays. Further study may indicate that more primitives are required to efficiently manipulate the arrays. This is particularly true in the case of uniformly structured general arrays (i.e. arrays where all the sub-arrays on a

level have identical structure). It should be possible to first implement these new functions as defined functions using the notation of this paper. For example a defined function to determine the depth of an array (i.e. the maximum number of levels as defined in chapter 1) could be written as follows:

```
      ΔDEPTH
[0]   D← ¯ A
[1]   →((⊂⍳0) ≡ ⍬⍴A)⍴D←0
[2]   →(D←(⍳0) ≡ ,A)/0
[3]   D←(¯ 1↓,A)⌈1+ ¯ ⊃0 ⌷ A
```

Type Determination

It is often useful for a function to adjust its action depending on the data type of arrays presented to it as operands. The inclusion of arrays consisting of different scalar types makes it difficult to test for the type of an array. It may therefore be necessary to provide a primitive for the purpose. Such a primitive could return a particular element of the type or perhaps merely an integer.

Multiple Functions

The synchronizing functions (wait,activate) proposed for multiple functions provide basic regulation of parallel

functions but have some shortcomings. It is not easy to cause a function to wait on multiple conditions. There is no primitive ability for one defined function to terminate, interrupt, or cause a branch in another defined function. Proper programming conventions, however, can provide these abilities.

Files

Perhaps the biggest deterrent to use of *APL* in many applications is the lack of a large data capability (a file system). This is largely an implementation problem and could be solved without modification to the notation by permitting (by declaration) large general arrays whose values were recorded external to the computer. Adding READ and WRITE functions to the notation in the spirit of traditional programming languages would not be an elegant solution.

System commands

Further study into the relation of the system commands of *APL*\360 to the notation is required. Perhaps they should be included formally in the language. Or perhaps the implementation environment should be altered so they are not needed at all.

D. Final Remarks

This paper has presented for consideration a generalization of the *APL* notation. The concepts developed extend and complement the existing capabilities of the notation. General arrays permit easy representation of data not amenable to rectangular structures (i.e. trees, lists). Including functions in the domain of variables makes the notation functionally self-extensible and makes consideration of *APL* for the native language of a computer more realistic.

It has been said (by Alan Perlis) that there is an inherent danger in extending *APL* in that the extensions may corrupt rather than improve. Whether or not the proposals made here are in good taste and in the spirit of *APL* in the ultimate analysis is in the hands of the users. The attempt has been made to preserve the essence of *APL* by preserving identities while extending the notation and the domain on which it is defined. It is hoped that the ideas presented here will stimulate further discussion and study of *APL*.

Appendix 1: Summary of Modifications

A. Syntax

The only change to the syntax of the notation is the requirement to parenthesize a niladic function used as the left operand of a function. The syntax is generalized in two ways:

1.) A function symbol or function name may be used in any of four contexts depending upon the existence of zero, one on the right, one on the left, or two operands. These types are called respectively niladic, monadic, dextri-monadic, and dyadic.

2.) Expressions having no value (no result after evaluation) may be embedded within expressions and may in a sense be left operands to niladic or monadic functions, or right operands to niladic and dextri-monadic functions.

The order of evaluation of functions is unchanged for expressions valid in *APL\360*. However the existence of embedded expressions of no value requires a more restrictive description of the order. A function occuring immediately to the right of a right parenthesis has lower precedence than

any function occuring in the parenthesized expression. Otherwise precedence is positional and increases from left to right. The evaluation rule then merely demands that functions be evaluated in order of decreasing precedence.

B. Arrays

The variety of arrays is increased by the inclusion of three new data types. The position scalar θ , is used as a placeholder in arrays when there is no reason to chose another scalar. A program scalar is the description of some function. A name scalar is an indirect reference to some value. Arrays whose scalars are of different types are permitted. An array which is composed of scalars is called a simple array. An array whose structure is hidden and which is treated as atomic is called a unit array. An array composed of unit arrays is called a general array. A general array is the regular, structured equivalent of a ragged array. An array whose rows of different length may be represented as a general vector whose elements are the rows of the ragged array.

C. Names

Names are treated more uniformly via the introduction of the variable function concept which treats both functions and variables as name-array pairs. A function is a name

associated with a program array, all other variable functions are called variables. Names may exist independently of arrays in the name domain. The name domain is recognized only in the left operand of specification (and in Chapter 3 in the left operand of activate). Specification may be used to define both functions and variables. Arrays of name scalars are produced by functions defined on names. These name arrays allow selective respecification of values of variables and allow splitting of a value among several names. Selective respecification is not new to *APL\360* since it exists under the guise of the triadic-looking indexed specification. Splitting of a value among names is new and is vital to the parameter substitution rules for defined functions.

D. Functions and Operators

A defined function is a name associated with a program scalar. An un-headed defined function is niladic and is a collection of associated expressions. A headed defined function is a collection of associated expressions the first of which is called the function header. The function header defines the valid context of the function, specifies the default operands, and declares sets of names as local names or strictly local names.

The following list of primitive functions are present in *APL\360* or are trivially extended to include new data types or general arrays. They may be located in the paper via the index.

Size ρR

Reshape $L\rho R$

Attach L,R

Base Value $L\perp R$

Represent $L\top R$

Compress L/R

Expand $L\backslash R$

Reverse ϕR

Rotate $L\phi R$

Membership $L\in R$

Transpose $\Diamond R$ and $L\ \Diamond R$

Grade Up ΔR

Grade Down ∇R

Branch $\rightarrow R$

The following operators are included as above

Outer Product $L\ \circ.D\ R$

Inner Product $L\ D.D'\ R$

The following functions exist in *APL\360* but have been altered or extended.

Index-of	$L\iota R$	array left operands
Interval	ιR	vector operand
Ravel	$,R$	function index permitted
Take	$L\uparrow R$	function index permitted
Drop	$L\downarrow R$	function index permitted
Specification	$L\leftarrow R$	left operand name arrays

The following operators are included as above

reduction	D/R	when unindexed, always produces a scalar. When indexed, any or all dimensions may be reduced
scan	$D\backslash R$	same as reduction

The following functions do not exist in *APL\360*

Conceal	$\subset R$	creates unit arrays
Reveal	$\supset R$	extracts array for a unit array
Unit Indexing	$L\ \exists\ R$	replaces bracket indexing
Entire	$\exists\ R$	an identity function
Same	$L = R$	=1 if operands are identical
Complement-of	$L\sim R$	set difference
Activate	$L\leftarrow$	create an active name

Define	∇R	create a program scalar
Evaluate	$\perp R$	evaluate an expression
Character form	τR	character representation
Exist	$\exists R$	=1 if name has a value
Default-of	$L \exists R$	defaults operands
Wait	$L \omega R$	suspends evaluation of expressions

The following operators do not exist in *APL\360*

Scalar Product	$\S D$	applies functions to concealed arrays
Definition-of	ΔR	produces an unevaluated variable description

Appendix 2: Prinitive Scalar Functions

D R	*RESULT*	*D*	*L D R*	*RESULT*
+*R*	*R*	+	*L*+*R*	*L* plus *R*
+2	2		3+2	5
-*R*	Minus *R*	-	*L*-*R*	*L* minus *R*
-2	¯2		4.5-1.5	3
×*R*	Signum of *R*	×	*L*×*R*	*L* times *R*
×¯1 0 3	¯1 0 1		4×4.5	18
÷*R*	Reciprocal of *R*	÷	*L*÷*R*	*L* divided by *R*
÷5	.2		4÷3	1.333333333
⌊*R*	Floor of *R*	⌊	*L*⌊*R*	Minimum of *L* and *R*
⌊3.14 ¯2.7	3 ¯3		3⌊4	3
⌈*R*	Ceiling *R*	⌈	*L*⌈*R*	Maximum of *L* and *R*
⌈3.14 ¯2.7	4 ¯2		3⌈4	4
**R*	e to the *R*th power	*	*L***R*	*L* to the *R*th power
*1	2.718281828		2*.5	1.414213562
⍟*R*	natural log of *R*	⍟	*L*⍟*R*	Log *R* to the base *L*
⍟**R* ↔ *R*	↔**⍟R*		*L*⍟*R* ↔ (⍟*R*)÷⍟*L*	
|*R*	Magnitude of *R*	|	*L*|*R*	*L* residue of *R*
|¯2.718	2.718		*L*≠0	*R*-*L*×⌊*R*÷*L*
			L=0	*R*
!*R*	Generalized factorial	!	*L*!*R*	*R* things *L* at a time
!*R* ↔ Gamma(*R*+1)			*L*!*R* ↔ (!*R*)÷(!*L*)×!*R*-*L*	
			2!5	10
?*R*	Random roll from ⍳*R*	?	(dyadic is non-scalar)	
○*R*	PI times *R*	○	*L*○*R*	*L*th circular function
○1	3.141592654			(see table on next page)
~*R*	Not *R*	~		
~1 ↔ 0	~0 ↔ 1			

		L	R	L∧R	L∨R	L⍲R	L⍱R
∧	And	0	0	0	0	1	1
∨	Or	0	1	0	1	1	0
⍲	Nand	1	0	0	1	1	0
⍱	Nor	1	1	1	1	0	0

<	Less
≤	Not greater
=	Equal
≥	Not less
>	Greater
≠	Not equal

RELATIONS

Result is 1 if the relation holds; 0 if it does not.

The Circular Functions

(-*L*)○*R*	*L*	*L*○*R*
(1-*R**2)*.5	0	(1-*R**2)*.5
Arcsin *R*	1	Sine *R*
Arccos *R*	2	Cosine *R*
Arctan *R*	3	Tangent *R*
(¯1+*R**2)*.5	4	(1+*R**2)*.5
Arcsinh *R*	5	Sinh *R*
Arccosh *R*	6	Cosh *R*
Arctanh *R*	7	Tanh *R*

Identity Elements

D	*ELEMENT*	
×	1	
+	0	
÷	1	(right only)
-	0	(right only)
*	1	(right only)
|	0	(left only)
⍟		(none)
○		(none)
∨	0	
∧	1	
⍲		(none)
⍱		(none)
!	1	(left only)
⌈		(smallest number representable)
⌊		(largest number representable)
>	0	(right only)
≥	1	(right only)
<	0	(left only)
≤	1	(left only)
=	1	
≠	0	

Appendix 3: Generating Well Formed Expressions

The following rules may be used for synthesis of partially parenthesized expressions with functions classified.

Non-terminal Symbols

EXP - an expression

EX0 - an expression with no value

EX1 - an expression with a value

E0 - an expression with no value and no operands

E1 - an expression with a value and no operands

ER0 - an expression with no value and a right operand

ER1 - an expression with a value and a right operand

EL0 - an expression with no value and a left operand

EL1 - an expression with a value and a left operand

ELR0 - an expression with no value and two operands

ELR1 - an expression with a value and two operands

LE1 - an expression as left operand of a function

FN0 - a niladic function with no result

FN1 - a niladic function with a result

FM0 - a monadic function with no result

FM1 - a monadic function with a result

FX0 - a dextri-monadic function with no result

FX1 - a dextri-monadic function with a result

FD0 - a dyadic function with no result

FD1 - a dyadic function with a result

Terminal Symbols

()[ ]⍵⍺ as before

*N*0 a niladic function with no result

*N*1 a niladic function with a result

*M*0 a monadic function with no result

*M*1 a monadic function with a result

*X*0 a dextri-monadic function with no result

*X*1 a dextri-monadic function with a result

*D*0 a dyadic function with no result

*D*1 a dyadic function with a result

The Rules

EXP -the starting expression

(1-3) EXP → EX0 | EX1 | (EXP)

(4-8) EX0 → ⍺ | E0 | ER0 | EL0 | ELR0

(9-11) EX0 → (EX0)EX0 | EX0(EX0) | (EX0)

(12-15) EX1 → E1 | ER1 | EL1 | ELR1

(16-18) EX1 → EX1(EX0) | (EX0)EX1 | (EX1)

(19-22) E0 → E0 E0 | ER0 EL0 | FN0 | (E0)

(23-26) EL0 → EL0 EL0 | ELR0 EL0 | LE1 FX0 | (EL0)

(27-29) ER0 → ER0 ER0 | ER0 ELR0 | FM0 ER1

(30-31) ER0 → FM0 ELR1 | (ER0)

(32-34) ELR0 → ELR0 ELR0 | EL0 ER0 | EL0 ELR0

(35-36) ELR0 → ELR0 ER0 | LE1 FD0 ER1

(37-38) ELR0 → LE1 FD0 ELR1 | (ELR0)

(39-42) E1 → E1 E0 | ER1 EL0 | ER0 EL1 | FN1

(43-45) EL1 → ELR1 EL0 | ELR0 EL1 | EL1 EL0

(46) EL1 → LE1 FX0

(47-50) ER1 → ER0 ELR1 | ER1 ELR0 | FM1 ER1 | FM1 ELR1

(51-53) ELR1 → ELR0 LLR1 | ELR1 ELR0 | EL0 ER1

(54-57) LLR1 → EL1 ER0 | (ELR1) | LE1 | LE1 FD1 ER1

(58) ELR1 → LE1 FD1 ELR1

(59-60) LE1 → (EX1) | *V*

(61-62) FN0 → *N*0 | *N*0[EXP]

(63-64) FN1 → *N*1 | *N*1[EXP]

(65-66) FM0 → *M*0 | *M*0[EXP]

(67-68) FM1 → *M*1 | *M*1[EXP]

(69-70) FX0 → *X*0 | *X*0[EXP]

(71-72) FX1 → *X*1 | *X*1[EXP]

(73-74) FD0 → *D*0 | *D*0[EXP]

(75-76) FD1 → *D*1 | *D*1[EXP]

Example derivations and classifications giving the rule numbers

*V F*1 *V*

EXP

2 → EX1

15 → ELR1

58 → LE1 FD1 ELR1

56 → LE1 FD1 LE1

60 → LE1 FD1 *V*

75 → LE1 *D*1 *V*

60 → *V D*1 *V*

(*V F*0)*F*0

EXP

1 → EX0

9 → (EX0) EX0

5 → (EX0) E0

21 → (EX0) FN0

61 → (EX0) *N*0

7 → (EL0) *N*0

25 → (LE1 FX0) *N*0

69 → (LE1 *X*0) *N*0

60 → (*V X*0) *N*0

*F*1 *F*0 *V* *V*

EXP

2 → EX1

13 → ER1

49 → FM1 ER1

47 → FM1 ER0 ELR1

56 → FM1 ER0 LE1

60 → FM1 ER0 *V*

30 → FM1 FM0 ELR1 *V*

56 → FM1 FM0 LE1 *V*

60 → FM1 FM0 *V* *V*

65 → FM1 *M*0 *V* *V*

67 → *M*1 *M*0 *V* *V*

(*V F*0 *V*) *F*1 *V*

EXP

2 → EX1

17 → (EX0) EX1

13 → (EX0) ER1

50 → (EX0) FM1 ELR1

56 → (EX0) FM1 LE1

60 → (EX0) FM1 *V*

67 → (EX0) *M*1 *V*

8 → (ELR0) *M*1 *V*

37 → (LE1 FD0 ELR1) *M*1 *V*

56 → (LE1 FD0 LE1) *M*1 *V*

60 → (LE1 FD0 *V*) *M*1 *V*

73 → (LE1 *D*0 *V*) *M*1 *V*

60 → (*V D*0 *V*) *M*1 *V*

References

1. Abrams,P.S.[1970]. An *APL* Machine. Report No. 114 Stanford Linear Accelerator Center, Stanford University (February) AD706 741

2. Breed,L.M.[1971]. Correspondence, *APL* Quote Quad Vol. 2, No. 6 (March)

3. Breed,L.M.[1971]. "Generalizing *APL* Scalar Extension". *APL* Quote Quad Vol. 2,No. 6 (March)

4. Charmonman,S.[1970]. "A Generalization of *APL* Array-oriented Concept". *APL* Quote Quad Vol 2, No. 3 (September)

5. Dijkstra,E.W.[1968]. "Cooperating Sequential Processes" Programming Languages, Academic Press, New York, 52-76

6. Edwards,E.M.[1971] Personal correspondence, Control Data Canada

7. Falkoff,A.D.,Iverson,K,E, and Sussenguth,E.H.[1964]. "A Formal Description of SYSTEM/360". IBM Systems Journal Vol. 3, No. 3, 198-262

8. Gilman,L.,Rose,A.J.[1970]. *APL*/360 An Interactive Approach, John Wiley and Sons, New York

9. IBM Corporation [1969]. *APL*\360 Users Manual, Form No. GH20-0683-0, White Plains, New York

10. Iverson,K.E.[1962]. A Programming Language, John Wiley and Sons, New York

11. Lathwell,R.H. and Mezei,J.E.[1971]. A Formal Description of *APL*\360, Draft of a Philadelphia Scientific Center Report (March)

12. Lathwell,R.H., Personal correspondence, IBM Philadelphia Scientific Center

13. Leeson,D.N. and Dimitry,D.L.[1962]. Basic Programming Concepts and the IBM 1620 Computer, Holt, Rinehart and Winston, New York

14. Liu,Y.[1971]. "Reverse Operator in *APL*", Computing Center News Vol. 4, No. 4, Syracuse University (March)

15. McDonnell,E.E., Personal correspondence, IBM Philadelphia Scientific Center

16. McCarthy,J.,et.al.[1962]. LISP 1.5 Programmer's Manual, M.I.T. Press

17. Morrow,L.A., Personal correspondence, IBM Philadelphia Scientific Center

18. Pakin,S.[1968]. *APL*\360 Reference Manual, Science Research Associates, Inc., Chicago, Illinois

19. Rubin,W.B.,Personal correspondence, Syracuse University

20. Ryan,J.[1971]. Generalized Lists and Other Extensions, *APL* Quote Quad Vol 3,No. 1 (June)

21. Watson,D.,McEwan,A.[1970]. "*APL*\360 Recursed", *APL* Quote Quad Vol. 2, No. 2 (July)

22. Woodrum,L.J.[1969]. "Internal Sorting with Minimal Comparing". IBM Systems Journal Vol 8,No. 3 189-219

INDEX OF IDENTITIES

INDEX

BIOGRAPHICAL DATA

Name: James Arthur Brown

Date and Place of Birth: December 23, 1943
Erie, Pennsylvania

Elementary School: Jefferson School, Erie Pennsylvania;
St. Peter's School, Erie Pennsylvania
Graduated 1957

High School: Cathedral Preparatory School
Erie, Pennsylvania
Graduated 1961

College: Gannon College, Erie, Pennsylvania
B.A., 1965

Graduate Work: Syracuse University, Syracuse, New York
Graduate Assistant
M.S., 1970